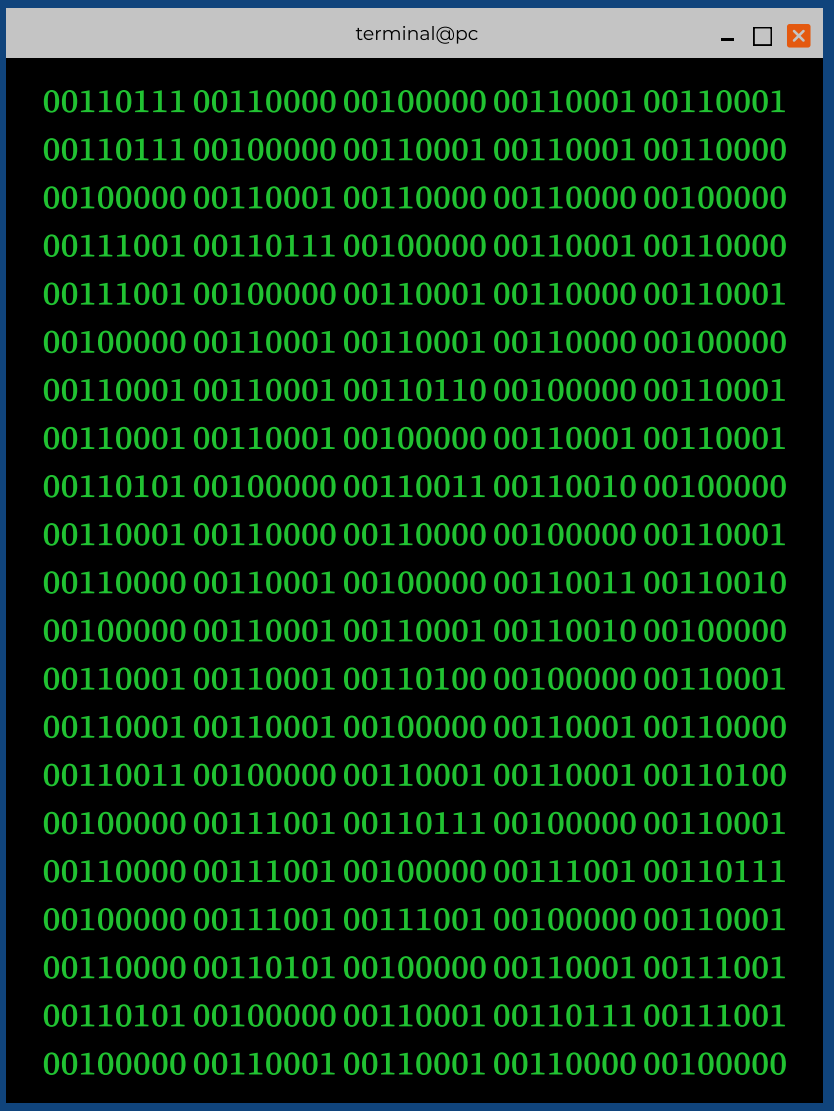
FUNDAMENTOS DE PROGRAMACIÓN



Eddy Ramírez-Jiménez

Índice general

Introducción

> La educación produce un cambio esencial en el hombre, o no es educación del todo
>
> Roberto Brenes Mesén

Mucho se ha hablado de la importancia de aprender a programar a todas las edades, recientemente Barack Obama se convirtió en el primer presidente estadounidense en hacer un programa, algunas universidades se han dedicado a hacer lenguajes de programación para niños y en algunos países están enseñando conceptos básicos de programación a los niños de escuela o kinder. Lo que nos lleva a pensar que este interés generalizado por aprender a programar se trata de lograr en los estudiatnes habilidades que antes no tenían y que se considera que la programación es la herramienta apropiada para tales efectos.

En opinión del autor, la programación es la herramienta que obliga al estudiante a abstraer generalizaciones de conceptos y métodos, de modo que pueda describir con profundidad un proceso. Este hecho, cambia la manera de pensar, observar el entorno, "abrir" la mente y por ende la capacidad de una persona para ver y comprender el mundo.

Los temas incluídos en este libro buscan ser una introducción para un curso de programación. Cada capítulo tiene una base teórica pequeña y un banco de preguntas dirigido a estudiantes que inician su carrera de ingeniería en sistemas, computación o ciencias de la computación por igual.[1]

Una característica de este libro es que carece de un lenguaje de programación específico dentro del texto, sino que está centrado principalmente en la algoritmia, esto con el fin de permitir que cada centro de estudios o estudiante pueda elegir el lenguaje que mejor le parezca; sin embargo, el autor recomienda fuertemente el uso de Scheme o Erlang dada su simplicidad en lo que respecta a sintaxis y a que abstrae muchos de los problemas de otros lenguajes en lo referente al manejo de tipos, que principalmente responden a características de las arquitecturas de las computadoras. Sin embargo, para facilitar la comprensión de estos puntos, el libro cuenta con un apéndice el cual ofrece una introducción

[1] Se espera que el lector tenga un dominio de algunos conceptos básicos como funciones, ecuaciones, variable, grado de un monomio, grado de un polinomio, suma, resta, multiplicación y división aritmética y de polinomios.

con la solución de los problemas de algunos ejercicios en seis lenguajes: C, Java, Scheme, Erlang, Prolog y Python.

La estructura del libro está dividida de la siguiente manera:

1. *Capítulo 1*: Este capítulo señala algunos principios matemáticos básicos que son necesarios para poder comprender y desarrollar algunos de los ejercicios planteados en el resto del libro. Se considera sólo un material de apoyo para los conocimientos que un estudiante de secundaria debe dominar en el área de aritmética, teoría de números, conjuntos y funciones.

2. *Capítulo 2*: Este capítulo explica lo que es una computadora digital, sin querer profundizar mucho en el contexto, puesto que este no es un libro de arquitectura de computadoras, pero el capítulo sienta las bases para poder comprender como es que la computadora logra ejecutar programas. Tanto el capítulo 2 como el capítulo 1 son parte de este libro para dar un contexto general de lo que es la programación. El resto de los capítulos tienen temas más clásicos de la programación de computadoras.

3. *Capítulo3:* Como todo componente básico de programación, se debe contar con conceptos fundamentales sobre funciones y fórmulas matemáticas básicas. Este capítulo explica conceptos claves y contiene problemas clásicos con los respectivos pasos para resolverlos. Se espera que el estudiante pueda apreciar la rapidez de cálculo que una computadora posee.

4. *Capítulo4:* Una de las principales habilidades que todo programador debe poseer, es el manejo de la **recursión** como herramienta de programación, esto debido al gran número de problemas que se pueden resolver por este mecanismo. El capítulo brindará problemas clásicos sobre recursión, además se incluye una defínción formal del concepto.

5. *Capítulo5:* Cuando un programador ya posee bases de programación, el siguiente paso es que haga conciencia sobre cómo programar bien. Este capítulo de eficiencia intenta hacer hincapié en el número de pasos que se necesita para completar un algoritmo, sin hacer un análisis formal sobre lo que es notación asintótica.

6. *Capítulo6:* En este capítulo se analizan y se ven diferentes operaciones sobre listas de números principalmente. También se ven algunos conceptos de álgebra para poder realizar ejercicios con listas considerándolas vectores y matrices.

7. *Capítulo7:* Sin pretender ser un libro de estructuras de datos, el capítulo de pilas, colas y árboles binarios busca ser una breve introducción a los problemas que conducen a este tipo de estructuras.

8. *Capítulo8:* Este capítulo final pretende mostrar al lector diversos problemas que se deben enfrentar día con día en programación para mostrar varios caminos que pueden seguirse en caso de querer ahondar en algún

área particular de la computación. Si el lector ha encontrado placer en la programación, muchos de los problemas finales de este capítulo los va a encontrar particularmente entretenidos y retadores.

Este libro también cuenta con una serie de apéndices, incluso el lector quizá quiera comenzar por alguno de ellos en particular. La razón por la que son apéndices y no capítulos del libro es porque el libro está centrado en programación, mientras que los apéndices buscan brindar herramientas para trabajar en ambientes de programación. A continuación, una breve descripción de cada uno de los apéndices:

1. Apéndice A: Este apéndice trata sobre los diferentes paradigmas de programación. Cada paradigma tiene sus propiedades y formas diferentes de abordar los problemas.

2. Apéndice B: Este apéndice da un poco de contexto histórico y semántico de varios lenguajes que por sus características son considerados emblemas, de una u otra forma, en cada uno de sus paradigmas. Los lenguajes son: *C*,*Prolog*,*Scheme*,*Erlang*, *Java* y *Python*.

3. Apéndice C: Este apéndice muestra la solución a los ejercicios del capítulo 3 en cada uno de los lenguajes. En el caso de los lenguajes cuya naturaleza es meramente recursiva (*Scheme*, *Erlang*, *Prolog*, *C*, *Java* y *Python*) pues también resuelven algunos del capítulo 4. Este apéndice debe manejarse con cuidado por parte del lector, se recomienda primero intentar hacer los ejercicios por sí mismos antes de ver las soluciones.

Capítulo 1

Fundamentos de Matemática

> Dos cosas contribuyen a avanzar: ir más deprisa que los otros o ir por el buen camino
>
> René Descartes

> La matemática es la ciencia del orden y la medida, de bellas cadenas de razonamientos, todos sencillos y fáciles
>
> Ídem

> Usted tiene perfecto derecho a elegir entre conocer las matemáticas o no, pero debe ser consciente de que, en caso de no conocerlas, podrá ser manipulado más fácilmente
>
> John A. Paulos

Este capítulo consiste en un repaso de algunos conceptos que se ven en cursos normales de primaria y secundaria, basados principalmente en lo que es teoría de números y lógica, la cual es vital y muy necesaria para poder comprender todo lo referente a programación.

Si el lector o lectora considera que sus bases matemáticas son los suficientemente sólidas, quizá desee sólo leer este capítulo y pasar por alto la realización de los ejercicios propuestos.

1.1. Fundamentos básicos de lógica

La lógica es importante porque constituye un área de la matemática cuyo estudio es el encargado de binrdar las explicaciones del por qué las cosas "funcionan" como lo hacen. Podemos sobre ella hacer, diferentes conjeturas, inferencias y teoremas que nos permiten avanzar más en el descubrimiento (o según algunos autores, invención) de las matemática.

La lógica es la parte de la matemática que se encarga de asignar valores de verdad a diferentes proposiciones. Una proposición es una expresión que puede ser catalogada como cierta o falsa, de ahí que tenga un valor de verdad.

Ejemplos de proposiciones:

- $5 < 2$ la cual es *falsa*
- $20 = 30$ la cual es *falsa*
- $30 = 3 \times 10$ la cual es *cierta*
- *El Sol emite luz de color blanca* la cual es *cierta*

Tradicionalmente, para su estudio, las proposiciones son nombradas con una letra mayúscula del alftabeto. Por ejemplo: P, Q, R, S, etc.

De esta manera podemos asignar una expresión con un nombre arbitrario: Ejemplo: "El Sol emite luz roja" $\equiv P$.

1.1.1. Operadores de proposiciones

Las proposiciones se pueden operar y se puede obtener su valor de verdad conociendo el valor de verdad de las proposiciones originales. Puede que sean verdaderas (designado como V) o falsas (designado como F).

En las siguientes subsecciones se indican los operadores lógicos básicos.

No - Negación

El operador *no* u operador de negación es el único que es unario (se aplica sobre una sola proposición, no requiere de más proposiciones) se representa con el símbolo $\sim$. A continuación se muestra un ejemplo: "El Sol ***no*** emite luz roja" $= \sim P$[1], recalcar que la negación de una proposición es representada con el símbolo $\sim$ antes de la proposición.[2]

Exite una tabla de verdad asociada[3], la cual se muestra a continuación:

P	$\sim P$
V	F
F	V

[1] Donde P es aún, "El Sol es rojo"

[2] Es posible que en algunos textos utilicen el símbolo $\neg$ para la negación

[3] Una tabla de verdad es una tabla donde se muestran todas las posibles combinaciones de los valores de verdad de las k proposiciones involucradas, por lo que debe poseer 2^k filas.

Comprender esta tabla es un ejercicio muy sencillo. La primera línea nos indica que P es verdadero, por lo que la negación de eso que es verdadero resulta falso. La segunda línea indica que P es falso, negarlo, resulta en una verdad.

Y - Conjunnción

El operador *y* u operador de conjunción, es un operador binario (requiere de dos proposiciones) el cual se representa con el símbolo $\wedge$. Suponga que tiene dos proposiciones:
$P \equiv$ Llevo pan
$Q \equiv$ Llevo queso
Para poder indicar la conjunción de ambas ("Llevo pan y llevo queso") se denotaría como $P \wedge Q$. En principio este operador indica que ***todas*** las proposiciones son ciertas, para que la expresión total sea cierta.

Para comprender el operador, se muestra su tabla de verdad correspondiente, dado que se involucran dos proposiciones, la tabla tiene 4 filas.

P	Q	$P \wedge Q$
V	V	V
V	F	F
F	V	F
F	F	F

La tabla debe interpretarse de la siguiente manera, volviendo al ejemplo donde $P \equiv$ Llevo pan y $Q \equiv$ Llevo queso, es necesario indicar para cada posible combinación de verdadero y falso, el resultado de la operación $P \wedge Q$.

La primera línea nos indica que es cierto que *Llevo pan*, también es cierto que *Llevo queso*, por lo tanto, cuando se indica "Llevo pan y llevo queso" resulta cierto.

La segunda línea indica que es cierto que *Llevo pan*, **no** es cierto que *Llevo queso*, por ende *Llevo pan y llevo queso* **no** es cierto.

La tercera línea es análoga a la anterior, pero a la inversa, lo que no se lleva es pan. La cuarta línea indica que no se lleva ninguna de las dos, por lo cual tampoco es cierto que lleve las dos.

O - Disjunción

El operador *o* u operador de disjunción, es un operador binario (requiere de dos proposiciones) el cual se representa con el símbolo $\vee$. En general, el operador da un resultado verdadero cuando ***alguna*** de las proposiciones es cierta.

Suponga que tiene dos proposiciones:
$P \equiv$ Llevo pan
$Q \equiv$ Llevo queso
Para poder indicar que "Llevo pan o llevo queso" usaría $P \vee Q$. En principio, este operador indica que alguna de las proposiciones es cierta, para que la expresión completa sea cierta.

Para comprender el operador, se muestra su tabla de verdad correspondiente, dado que se involucran dos proposiciones, la tabla tiene 4 filas.

P	Q	$P \vee Q$
V	V	V
V	F	V
F	V	V
F	F	F

La tabla debe interpretarse de la siguiente manera, volviendo al ejemplo donde $P \equiv$ Llevo pan y $Q \equiv$ Llevo queso, es necesario indicar cuando es que $P \vee Q$ es verdad o una mentira.

La primera línea nos indica que es cierto que *Llevo pan*, es cierto que *Llevo queso*, por lo tanto, cuando se indica "Llevo pan o queso" también es cierto.

La segunda línea indica que es cierto que *Llevo pan*, **no** es cierto que *Llevo queso*, por ende *Llevo pan o queso* **e**s cierto, dado que se requiere que sólo uno de las proposiciones sea cierta.

La tercera línea es análoga a la anterior, pero a la inversa, lo que no se lleva es pan.

La cuarta línea indica que no se lleva ninguna de las dos, por lo cual tampoco es cierto que lleve alguna de las dos.

Implica

Este operador es un operador que tiene como principio, P implica Q, es decir, P es una causa y Q es una consecuencia. Se denota con el símbolo $\implies$. En lenguaje natural, la expresión sería "Si (P) entonces (Q)".

La tabla de verdad se muestra a continuación:

P	Q	$P \implies Q$
V	V	V
V	F	F
F	V	V
F	F	V

Es importante recordar que lo que la tabla indica es cuando $P \to Q$ es mentira y si no es mentira, por lo tanto es verdad.[4].

Para comprender mejor el ejemplo, imaginemos que un profesor suyo le hace la siguiente afirmación: *Si viene a todas las clases,entonces le pongo 100 en todos los quices.* A partir de esa expresión, se puede traducir como:
$P \equiv$ viene a todas las clases
$Q \equiv$ le pongo 100 en todos los quices.

Veamos un análisis fila por fila:

[4]Véase la tabla de verdad de la negación

1. $P \equiv V$ $Q \equiv V$

   P es verdad (usted llegó a todas las clases)

   Q es verdad (el profesor le puso 100 en los quices)

   $P \implies Q$ es verdad, el profesor no mintió

2. $P \equiv V$ $Q \equiv F$

   P es verdad (usted llegó a todas las clases)

   Q es falso (el profesor no le puso 100 en los quices)

   $P \implies Q$ es falso, el profesor mintió

3. $P \equiv F$ $Q \equiv V$

   P es falso (usted no llegó a todas las clases)

   Q es verdad (el profesor le puso 100 en los quices)

   $P \implies Q$ es verdad, el profesor no mintió (le dijo lo que pasaba si llegaba, nunca le dijo lo que pasaba si no llegaba)

4. $P \equiv F$ $Q \equiv F$

   P es falso (usted no llegó a todas las clases)

   Q es falso (el profesor no le puso 100 en los quices)

   $P \implies Q$ es verdad, el profesor no mintió

O-exclusivo

El o-exclusivo sirve para indicar cuando las proposiciones son excluyentes entre sí, por ejemplo $P(x) \equiv$ x es alto y $Q(x) \equiv$ x es bajo. Nótese que no son necesariamente opuestos, hay puntos intermedios, de personas que no son ni altas ni bajas, pero está claro que no se puede ser alto y bajo simultáneamente. El símbolo es $\veebar$

P	Q	$P \veebar Q$
V	V	F
V	F	V
F	V	V
F	F	F

Como se puede apreciar, a través de la tabla, cuando ambos son ciertos, entonces es falso, precisamente porque es imposible. Fuera de ese detalle, el comportamiento es igual que el de una disjunción normal (O).

La comprensión de la lógica es fundamental para comprender cambios de flujo en la ejecución de algoritmos, los cuales se verán más adelante, también es ampliamente utilizado en el diseño digital, base de la computación actual.

1.2. Conjuntos

Los conjuntos son grupos de elementos que cumplen con una condición lógica particular. Estos conjuntos se pueden definir en infinitos conjuntos, por ejemplo, algunos de ellos pueden ser:

- {x / x es uno de los tres chiflados}

  Para indicar que se trata del famoso grupo de comediantes

- {x / x es un animal mamífero }

  Este otro caso indica que es el conjunto formado por los animales mamíferos

- {x/ x es uno de los conjuntos en los ejemplos anteriores}

  Este conjunto está formado por sólo dos elementos, el primero es el conjunto de los tres chiflados y el segundo es el conjunto de los mamíferos. La mejor forma de entender esta situación, es imaginar los conjuntos como bolsas virtuales que dentro pueden tener otras bolsas, de esta manera, esta última bolsa tiene únicamente dos bolsas, que dentro de estas dos bolsas haya más elementos, es absolutamente independiente.

- $\emptyset$

  Este otro es un caso particular de un conjunto que no tiene elementos, para continuar con la metáfora, es una bolsa vacía, de ahí que se titule conjunto vacío.

Un detalle importante es que un conjunto no puede tener elementos repetidos, es decir todos los elementos deben ser diferentes entre sí.

Para identificarlos y no tener que describir a los conjuntos todo el tiempo, se les asigna nombres. Esto es algo muy útil para poder identificar más rápidamente los conjuntos. Por ejemplo puedo hablar del conjunto de los *mamíferos* sin tener que describir que son *aquellos animales vertebrados que producen leche para sus recién nacidos.*

En matemática se utilizan típicamente letras mayúsculas del alfabeto occidental a partir de la A (A,B,C,D...) cuando el conjunto está siendo definido por proposiciones particulares (ej: Sea A el conjunto de los números enteros pares mayores a cero), se usa de manera genérica (ej: sea A un conjunto finito) o se describe completamente (ej: sea A = {1,2,3}).

1.2.1. Cuantificadores

Para indicar que ciertos elementos de un conjunto cumplen ciertas propiedades adicionales, se utilizan ciertos símbolos para indicar si "algunos" lo cumplen o si "todos" lo cumplen.

Para suplir esta necesidad, se cuenta con el cuantificador $\exists$ que quiere decir existe y con el cuantificador $\forall$ que quiere decir todos.

Si se quiere decir que, por ejemplo, *"algunos mamíferos ponen huevos"*. Indicando que M quiere decir el conjunto de los mamíferos y $H(x)$ que x pone huevos, puedo denotarlo así: $\exists x \in M/H(x)$ (En este caso particular x haría referencia al equidna y al ornitorrinco).

Si por el contrario, quisiera expresar *"todo cuerpo sumergido dentro de un fluido experimenta una fuerza ascendente llamada empuje, equivalente al peso del fluido desalojado por el cuerpo"*[5], una vez más, puedo indicar que S es el conjunto de los cuerpos que se sumergen en un líquido y $F(x)$ x sufre una fuerza de empuje ascendente llamada empuje equivalente al peso del fluido desalojado por x, podría entonces denotarse de la siguiente forma: $\forall x \in S/F(x)$.

Finalmente, hay un caso especial que es el "existe un único" el cual se utiliza cuando exactamente uno de los elementos del conjunto lo cumple, por ejemplo: $V(x)$ quiere decir que existe vida comprobada en x, y P el conjunto de los 8 planetas del Sistema Solar, puedo decir que $\exists! x \in P/V(x)$.

1.2.2. Relaciones entre conjuntos

Los conjuntos se conforman por elementos. Para indicar que un elemento x es parte de un conjunto se denota con el símbolo $\in$ el cual se lee: "es elemento de" o "pertenece a", de esta forma podemos indicar si un elemento es parte o no de un conjunto. Ejemplo: $\pi \in \mathbb{R}$ o cuando un elemento **no** pertenece, que sería mediante $\notin$, ejemplo$\pi \notin \mathbb{N}$

Cuando todos los elementos de un conjunto A son elementos de otro conjunto B ($\forall x \in A, x \in B$) se dice que A es *subconjunto* de B y se denota con $\subseteq$, de forma que si $\forall x \in A, x \in B \implies A \subseteq B$.

Nótese que la definición anterior no excluye a $A = B$, si se quisiera indicar que $A \neq B$, entonces A es un *subconjunto propio* de B y se denota por $\subset$ que quiere decir que todos los elementos de A están en B, pero existe al menos un elemento de B que no es parte de A ($\forall x \in A/x \in B \wedge \exists y B/y \notin A$). De igual manera para indicar que hay un elemento de A que no es parte de B, se denota con una raya en medio del símbolo $\not\subset$, que quiere decir $\exists x \in A/x \notin B \implies A \not\subset B$.

1.2.3. Operaciones de conjuntos

Existen varias operaciones que se pueden realizar entre los conjuntos. A continuación se describen cada uno de ellos. Los ejemplos están dados suponiendo los siguientes conjuntos: $A = \{1, 2\}$ y $B = \{2, 3\}$

1. $\cup$: Unión

   Es el conjunto resultante de tomar todos los elementos que están en uno u otro conjunto.

   $A \cup B = \{x/x \in A \vee x \in B\}$

   $A \cup B = \{1, 2, 3\}$

[5] Principio de arquímides

2. $\cap$: Intersección

   Es el conjunto resultante de tomar los elementos en común para los dos conjuntos.

   $A \cap B = \{x/x \in A \wedge x \in B\}$

   $A \cap B = \{2\}$

3. $-$: Resta o Diferencia simple

   Es el conjunto resultante de tomar todos los elementos del primer conjunto que no están en el segundo.

   $A - B = \{x/x \in A \wedge x \notin B\}$

   $A - B = \{1\}$

4. $\triangle$: Diferencia Simétrica

   Es el conjunto resultante de tomar todos los elementos que están en exactamente uno de los dos conjuntos, pero no en los dos. Es decir, la unión menos la intersección.

   $A \triangle B = \{x/x \in A \veebar x \in B\}$

   $A \triangle B = \{1, 3\}$

5. $\times$: Producto Cartesiano

   Es el conjunto formado por pares, donde el primer ítem de cada par es un elemento del primer conjunto y su segundo ítem es del segundo conjunto

   $A \times B = \{(x, y)/x \in A \wedge y \in B\}$

   $A \times B = \{(1, 2), (1, 3), (2, 2), (2, 3)\}$

1.2.4. Conjuntos de números

Dado que los números es lo más tratado en matemática, a muchos conjuntos de ellos se les ha asignado letras . Tradicionalmente indicadas por una doble línea en algunas de sus partes y siempre utilizando letras en mayúscula($\mathbb{A}, \mathbb{B}, \mathbb{C}, \mathbb{D}, \mathbb{E}, \mathbb{F}, \mathbb{G}, \mathbb{H}, \mathbb{I}, \mathbb{J}, \mathbb{K}, \mathbb{L}, \mathbb{M}, \mathbb{N}, \mathbb{O}, \mathbb{P}, \mathbb{Q}, \mathbb{R}, \mathbb{S}, \mathbb{T}, \mathbb{U}, \mathbb{V}, \mathbb{W}, \mathbb{X}, \mathbb{Y}, \mathbb{Z}$)[6].

La siguiente tabla menciona y describe los conjuntos más conocidos:

[6] No todos los conjuntos escritos están definidos, pero si se llegaran a definir, así se deberían representar

$\mathbb{R}$	Conjunto de los reales. Básicamente son los números que "existen".
$\mathbb{C}$	Estos son los complejos que en el fondo es el producto de un real por i
$\mathbb{I}$	Los irracionales, son aquellos números reales que no tienen período
$\mathbb{Q}$	Los racionales son los reales que sí tienen período
$\mathbb{Z}$	Los enteros son racionales que luego de la coma, tienen período de cero
$\mathbb{N}$	Los naturales son los enteros que son mayores o iguales a cero

Vale la pena señalar que $\mathbb{Z}$ está divido en tres conjuntos, el primero el de los positivos ($\mathbb{Z}^+$), el de los negativos ($\mathbb{Z}^-$) y el cero ($\{0\}$) el cual no es ni positivo ni negativo.

1.3. Relaciones binarias

Una relación binaria es aquella que se define sobre dos conjuntos (no necesariamente distintos) se trata de un conjunto a su vez de pares. Por ejemplo, sobre $\mathbb{A} = \{1, 2, 3, 4\}$ se define una relación R de la siguiente manera $\{(1,1), (1,2), (2,3), (4,1), (4,3), (3,2), (3,3)\}$

1.3.1. Propiedades de una relación

1. Reflexiva

   Una relación es reflexiva si todos los elementos están relacionados consigo mismos. Por ejemplo, una relación que incluya la igualdad.

2. Simétrica

   Una relación es simétrica si cada vez que (aRb) entonces (bRa), por ejemplo, la relación hermano, si A es hermano de B, entonces B también es hermano de A.

3. Transitiva

   Una relación es transitiva si $(aRb) \wedge (bRc)$ entonces (aRc). Por ejemplo, la relación descendiente, si A desciende de B y B desciende de C, entonces A también desciende de C.

4. Antisimétrica

   Una relación es antisimétrica si únicamente si $(a = b) \rightarrow aRb \wedge bRa$. Por ejemplo, la relación $\leq$ donde si $a \leq b \wedge b \leq a$ entonces $a = b$

5. Total

   Una relación es total, si para cualesquiera dos i, j elementos de un conjunto, existe la relación (i, j).

1.4. Funciones

Las funciones son un tipo particular de relación que se establece entre dos conjuntos llamados Dominio y Codominio, donde para cada elemento del dominio se le indica una forma de calcular los elementos del Codominio. Es importante que todos los elementos del dominio tengan un par en el Codominio, pero no es necesario que todos los elementos del codominio estén cubiertos por la función. Al subconjunto de llegada, es decir al subconjunto del codominio que sí está cubierto por la función, se le llama *Ámbito*.

De esta manera si se define $f : A \rightarrow B$, siendo A el Dominio y B el Codominio, f es función sii $\forall x \in A \implies f(x) \in B$. Típicamente se le asigna a los elementos del Dominio el nombre x y al ámbito y.

1.4.1. Inyectividad

Una función es inyectiva si para cada y_i del ámbito se le asigna únicamente una x_i, de forma que si $f(x_i) = f(x_j) \implies x_i = x_j$.

1.4.2. Sobreyectividad

La sobreyectividad es indicada si el codominio es igual que el ámbito. Esto quiere decir que todos los elementos del codominio forman parte de la función.

1.4.3. Biyectividad

Una función es biyectiva si es sobreyectiva e inyectiva al mismo tiempo.

Algunas propiedades de la biyección

Sea $f : A \rightarrow B$ una función biyectiva, entonces se cumple:

1. $|A| = |B|$

   Explicación: $\forall x \in A \exists! y \in B / f(x) = y$ eso quiere decir que todos los elementos de A son correspondidos por uno de B, por lo tanto, la cantidad de elementos entre los dos, es la misma.

2. $\exists f^{-1} : B \to A$

   Explicación: Si invertimos el codominio y lo convertimos en dominio y viceversa, sigue habiendo una función, debido a que todos los elementos del codominio tienen una correspondencia en el dominio. La función f^{-1} recibe el nombre de *función inversa*.

1.5. Teoría de Números (naturales)

Uno de los conjuntos sobre los cuales se han generado mayor cantidad de estudios por más tiempo es el conjunto de los números naturales, los cuales fueron empezados a tratar por los griegos. De ahí que tenemos varias propiedades, en esta sección exploraremos algunas de ellas.

1.5.1. Premisas

Todo principio matemático está basado en axiomas que son aquellas premisas de las cuales se parte para hacer las conclusiones posteriores. A continuación estas premisas:

1. Cero es un número

   El primer axioma me indica que el cero es un número

2. $P(n) \implies P(n+1)$

   El segundo axioma me indica que si n es un número, entonces $n+1$ lo es también

3. Principio del buen ordenamiento

   $\mathbb{N}$ ordenable:

   - Todo subconjunto de $\mathbb{N}$ tiene un elemento menor.
   - No existe el descenso infinito en $\mathbb{N}$.

Operaciones

Dentro del conjunto $\mathbb{N}$ se van a definir tres operaciones principales:

- Suma $+$
- Resta $-$
- Multiplicación $\times$

1.5.2. Divisibilidad

Entre los números enteros existe una relación que es la de divibisilidad. La notación más conocida es $b|c$ que se lee b ***divide a*** c e implica que b es divisor de c, o c es múltiplo de b.

La definición más sencilla sería: $a, b, c \in \mathbb{Z}\ b \neq 0 \implies (b|c \iff \exists n \in \mathbb{Z}|c = b \cdot n)$

Propiedades de la divisibilidad

La divisibilidad en los naturales es una relación de orden, a continuación el detalle de la demostración:

- Reflexiva $a|a\ \forall a \in \mathbb{N}$
  Prueba:
  $a = a \cdot u$ donde $u = 1$
- Transitiva $(a|b \wedge b|c) \implies a|c$
  Prueba:
  $(a|b \wedge b|c)$
  $\implies b = a \cdot n_1 \wedge c = b \cdot n_2$
  Reemplazando la primera ecuación en la segunda
  $\implies c = a \cdot n_1 \cdot n_2$
  $\implies a|c$
- Antisimétrica $a|b \wedge b|a \implies a = b$
  Prueba:
  $a|b \wedge b|a$
  $\implies b = a \cdot c_1 \wedge a = b \cdot c_2$
  Reemplazando la primera en la segunda
  $\implies a = a \cdot c_1 \cdot c_2$
  $\implies 1 = c_1 \cdot c_2$
  $\implies c_1 = c_2 = 1$

Propiedades adicionales

Existen algunas propiedades de la divisibilidad en $\mathbb{N}$ que deben ser tomados en cuenta y cuya demostración es sencilla

1. $1|n\ \forall n$
   Todos los números son divisibles por 1 Prueba: $1|n \iff \exists x/1x = n$
   $\implies x = n$
2. $n|0\ \forall n$
   Todos los números dividen al cero, por lo que el cero es par, el cero es múltiplo de 3, de 4, etc. Prueba: $n|0 \iff \exists x/nx = 0$
   $\implies x = 0$

3. $m > 0 \wedge n|m \implies n \leq m$

   En caso de que no sea 0, un divisor de m es menor o igual que m. Prueba: $n|m \iff \exists x \in \mathbb{N}/nx = m$, dado que el mínimo natural que no reduce a cero la multiplicación, es $1 \implies x \geq 1 \implies nx \geq n$
   $\implies m \geq n$ por transitividad.

4. $a|b \iff a|(b \cdot n)\ \forall n$

   Si un factor de un número, es factor de cualquier múltiplo de ese número Prueba: $a|b \iff \exists k \in \mathbb{N}/ak = b \implies akn = bn$ dado que $kn \in \mathbb{N} \implies a|bn$. Las siguientes proposiciones se le sugiere al lector que las desarrolle en su casa.

5. $a|b \wedge a|c \implies a|(b + c)$

   Si un número es factor de otros dos, es factor de su suma

6. $a|b \wedge a|c \implies a|(b - c)$

   Si un número es factor de otros dos, es factor de su diferencia

7. $a|b \wedge a|c \implies a^2|(b \cdot c)$

   Si un número es factor de otros dos, el cuadrado es factor de su producto

1.5.3. Teorema de la divisibilidad euclideana

Dados dos números naturales a y b, con $b \neq 0$, la división euclídea asocia un cociente q y un resto r, ambos números naturales, se verifican:

$a = bq + r\ 0 \leq r < b \implies$ Se garantiza que la pareja dada por (q,r) existe y es única. En otras palabras, para cualquier dividendo y divisor, existen un cociente y residuo únicos.

Demostración

Este teorema es particularmente útil, muy conocido, pero pocas veces se demuestra. Partamos del principio de la contradicción:

Supongamos que para (a, b) existen dos pares: (q_1, r_1) y (q_2, r_2) tal que $q_1 \neq q_2 \wedge r_1 \neq r_2$ con la satisfacción de estas

$$a = bq_1 + r_1$$

$$a = bq_2 + r_2$$

Dado que ambos son iguales, se puede afirmar:

$$bq_1 + r_1 = bq_2 + r_2$$

Dividiendo entre b a ambos lados:

$$q_1 + \frac{r_1}{b} = q_2 + \frac{r_2}{b}$$

Es importante recordar que r_1 y r_2 son menores que b, por ende las fracciones $\frac{r_1}{b}$ y $\frac{r_2}{b}$ son propias (menores a 1), con lo que la parte entera de ambas partes de la ecuación está dada por q_1 y q_2.

$$q_1 = q_2$$

Lo que a su vez implica que

$$\frac{r_1}{b} = \frac{r_2}{b}$$

y esto implica que

$$r_1 = r_2$$

. Lo que lleva a una contradicción, porque se partió que eran diferentes.

1.5.4. Números primos

Un número primo es aquel número que tiene sólo 2 divisores enteros, él mismo y el 1. (En principio, porque son las dos condiciones mínimas de divisibilidad ($1|n\forall$ y $n|n\forall n$). En otras palabras que no posee ningún divisor propio[7] mayor a 1.

Los números primos tienen numerosas aplicaciones en computación, esto debido a las propiedades que poseen. Por ejemplo, una de ellas consiste en que al no tener factores, es actualmente costoso, desde el punto de vista computacional, factorizar números de más de 1000 dígitos por lo que mucho de lo referente a firma digital o cifrado de datos actualmente está basado en este tipo de propiedades. Si lo consideran de esta forma, cada vez que se coloca el número de la tarjeta de crédito o se tiene una conversación privada con alguien a través del internet, se está confiando plenamente en los números primos.

1.6. Máximo Común Divisor

El máximo común divisor de dos números[8] a y b, como su nombre lo indica es el mayor divisor que divide a ambos y se denota por (a, b).

1.6.1. Común divisor

Un divisor d de n es aquel que $d|n$. Sean los conjuntos A y B con los divisores de a y b respectivamente ($A = d_1, d_2, \ldots, d_i$ $/\forall d \in Ae|a$ y $B = e_1, e_2, \ldots, e_i$ $/\forall e \in Ae|b$). Los divisores comunes serían aquellos que están en la intersección: $A \cap B$.

[7] Un divisor propio es aquel que es estrictamente menor al número en cuestión

[8] Es extendible a n números, pero para iniciar el concepto se partirá del caso de dos

1.6.2. Máximo de un conjunto

Para que exista un elemento máximo en un conjunto deben garantizarse dos cosas:

1. Que el conjunto sea finito o infinito decreciente
2. Que el conjunto no sea vacío

En el caso de los divisores de un número, se trata de aquellos números menores o iguales que él y mayores que cero, que lo dividen, por lo que son un conjunto finito, de ahí que su intersección es finita. Sin embargo debe haber un número al menos en común que divide a los dos números, cualesquiera que sean estos dos. Afortunamente, tenemos que el 1 divide a todos los números. Por lo que la intersección en el peor caso tiene un elemento, el 1. Véase sección 1.5.2 en la página 24 para más detalles.

1.6.3. Coprimos

Dos números son *coprimos* (también llamados *primos relativos*) si no tienen divisores en común entre ellos, más que el uno. En otras palabras si el máximo común divisor entre ellos es el 1. Por ejemplo: el 25 y el 54 son primos relativos, dado que no comparten divisores en común.

1.6.4. Congruencias

Tal y como se vio en la seccioń 1.5.3 de la página 25, para cualquier par (a, n) existen dos únicos números (q, r) / $a = qn + r \iff 0 \leq r < n$, donde r es el residuo.

Las congruencias o también denominados *módulos*, son aquellos números cuyo residuo es el mismo para un divisor n.

Una de las congruencias más típicamente utilizadas, es la congruencia con 2 que incluso tienen un nombre, de esta manera llamamos *pares* a aquellos números que tienen residuo 0 al dividir por dos, es decir son *congruentes con 0 módulo 2*. De igual manera tenemos a los *impares* los cuales son *congruentes con 1 módulo 2*. Esto se da porque el número 2 sólo permite tener dos residuos posibles, el 0 y el 1.

Sin embargo, si se tiene módulo 3, se permiten tres residuos posibles $\{0, 1, 2\}$ y de manera general, módulo n tendrá n residuos posibles $\{0, 1, \ldots, (n-1)\}$.

Las congruencias se definen por compartir el mismo residuo con un n dado. Por ejemplo, los números congruentes módulo 5, serán aquellos que comparten el mismo residuo al dividir por 5.

La notación más común es $\equiv$ para la congruencia y (n) para el módulo, así si se quiere indicar que 7 y 12 comparten el mismo residuo módulo 5, se escribe: $7 \equiv 12(5)$

1.7. Teoremas y conjeturas

En esta sección se enuncian diferentes teoremas o conjeturas. Los teoremas ya están demostrados y se sabe que son ciertos. Las conjeturas, por su parte, parece que son ciertas, pues a pesar de haber buscando muchos contraejemplos, no se ha demostrado su falsedad, pero tampoco se ha demostrado su veracidad para todo el conjunto de los números naturales aún. Para cada uno de los enunciados que se indican, expréselos en forma matemática utilizando conjuntos, módulos y los demás principios que se describieron previamente.

1.7.1. Teorema de Fermat

1. El pequeño teorema de Fermat indica:

   Si p es un número primo, entonces, para cada número natural a, con $a > 0$, coprimo con p , (a) elevado a la (p-1) es congruente con 1 módulo p.

2. El gran teorema de Fermat indica:

   Sean los enteros a, b, c, n y se plantea la ecuación a elevando a la n igual a la suma de b elevado a la n y c elevado a la n, esa ecuación no tiene solución entera para n mayor a dos.

1.7.2. Teorema de Carmichael

Hay dos posibles enunciados de este teorema que llevan el mismo nombre, el primero es referente a la sucesión de Fibonacci[9] Que indica que para $n > 12$ la sucesión en su enésimo término tiene un factor primo que no es factor de ningún número de fibonacci previo a él.

El segundo enunciado de Carmichael hace referencia a la *función de Carmichael* la cual indica que dado un entero positivo n, se define como el menor entero m tal que cumple: a elevado a la m es congruente con 1 módulo n.

Se denota con la letra griega *lambda*(λ).

1.7.3. Teorema de los cuatro cuadrados

Este teorema fue demostrado por Lagrange, matemático francés, por lo que previamente fue conocido como *conjetura de Bachet* antes de que fuera demostrada, e indica que todo número natural se puede expresar como la suma de cuatro números naturales al cuadrado. Por ejemplo: $310 = 17^2 + 4^2 + 2^2 + 1^2$
Como dato curioso, en 1834, Carl Gustav Jakob Jacobi demostró que existe una fórmula que permite determinar el número total de maneras en que un número

[9]La sucesión de Fibonacci, que lleva el nombre del matemático que la propuso. La sucesión es como se muestra en la siguiente tabla:

f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	...	f_n
0	1	1	2	3	5	8	13	21	34	...	$f_{n-1} + f_{n-2}$

entero positivo n dado puede representarse como la suma de cuatro cuadrados. Este número es ocho veces la suma de los divisores de n si n es impar y 24 veces la suma de los divisores impares de n si n es par.

1.7.4. Teorema: El menor factor propio mayor que 1 de n es primo

Este teorema indica que el menor de los factores, mayor que 1 de un cualquier número es primo.

Como extensión a este teorema, también se sabe que si un número no tiene factores propios menores o iguales a su raíz, es primo.

1.7.5. Teorema fundamental de la aritmética

El teorema fundamental de la aritmética indica que todo número se puede factorizar como un producto de números primos y que si ordenamos esta secuencia de primos, es una secuencia única.

1.7.6. Conjetura de Goldbach

Todo número par mayor que 2 puede escribirse como suma de dos números primos. - Goldbach.

Este es uno de los problemas abiertos más grandes del mundo, hasta el momento se han probado con números bastante grandes (10^{80}) y se sigue cumpliendo.

1.8. Problemas

1.8.1. Funciones

Sobre conjuntos infinitos:

1. Genere una función que no sea ni sobreyectiva ni inyectiva
2. Genere una función que sea sobreyectiva pero no inyectiva
3. Genere una función que sea inyectiva pero no sobreyectiva
4. Genere una función que sea biyectiva

1.8.2. Números poligonales

Son aquellos números que tienen forman un polígono de k lados. Así se tienen los triangulares, cuadrados, pentagonales, etc. En el siguiente espacio se encuentra un ejemplo con los números triangulares.

Números triangulares Son aquellos números que forman un triángulo, tal y como se aprecia en la siguiente figura, son el 1, 3, 6, 10, 15, 21...

```
          1
        2   3
      4    5   6
    7   8   9   10
  11  12  13  14  15
16  17  18  19  20  21
```

La fórmula del n-simo número triangular es $\frac{n(n+1)}{2}$

Encuentre la fórmula para los cuadrados, pentagonales, hexagonales y haga un dibujo similar al realizado para los números triangulares en cada uno de ellos, hasta al menos, el tercer cuadrado, el tercer pentagonal y el tercer hexagonal.

1.8.3. Criba de Eratóstenes

Hace aproximadamente 27 siglos, hubo un griego en Alejandría cuyo nombre era Eratóstenes, quien definió una forma para indicar si un número era primo o no. Su criba sigue siendo de mucha utilidad hoy en día. Investigue en qué consiste la criba y utilizando ese método en la siguiente tabla indique cuáles números son primos.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100	101	102	103	104	105
106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132	133	134	135
136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165
166	167	168	169	170	171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190	191	192	193	194	195
196	197	198	199	200	201	202	203	204	205	206	207	208	209	210
211	212	213	214	215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236	237	238	239	240

1.8.4. Los cuatro cuadrados

Lagrange demostró con el teorema de los cuatro cuadrados que para todo natural n existen 4 números naturales (en este caso, se incluye al cero) tal que elevados al cuadrado y sumados, dan n. Para los siguientes números indique cuáles son todos los sumandos a, b, c, d de modo que $a^2 + b^2 + c^2 + d^2 = n$. Ejemplo:

$$4 = 1^2 + 1^2 + 1^2 + 1^2$$

$\vee$

$$4 = 2^2 + 0^2 + 0^2 + 0^2$$

Los números por calcular son: $1, 2, 3, 16, 32, 100, 101$

1.9. Resumen

La lógica es una de las áreas principales de la matemática, está presente en casi todas las áreas. En computación tiene una referencia importante porque a partir de la definición de número, conjunto, estado, función, etc, genera áreas de estudio más puntuales que son los orígenes de la computación misma.

Los conceptos repasados aquí como inyectividad, sobreyectividad, por ejemplo, son la base de los sistemas de compresión o de seguridad informática que rige el mundo de la computación digital.

Estas herramientas además nos facilitan el poder escribir programas que se ejecuten apropiadamente en una computadora. Las computadoras, desde el punto de vista teórico son el producto de la aplicación de este conjunto de conceptos y unos cuantos más, luego, cuando se llevaron a la práctica, dieron pie a la construcción de máquinas basadas en conjuntos de estados que eran capaces de diferenciar un estado de otro mediante la asignación de un número que los identificase (gigantesco, pero un número al fin), con lo que conocemos hoy como las computadoras digitales.

En el capítulo 2 se va a explicar la utilización de estos y otros conceptos para poder crear las computadoras, así como las partes que las componen.

Capítulo 2

¿Qué es una computadora?

Hardware, las partes de la computadora que pueden ser pateadas

Jeff Pesis

Nunca confíes en una computadora que no puede ser lanzada por la ventana

Steve Wozniak

Originalmente este capítulo no era parte del libro, sin embargo, es imposible lograr una noción completa de lo que es la programación, sin comprender en un nivel básico lo que es una computadora.

En este capítulo 2 primeramente se explicarán dos conceptos de vital importancia como lo son máquinas de estados finitos y sistemas numéricos, luego se introducirán algunos conceptos más propios de las computadoras actuales. Se hará especial hincapié en el procesador.

Para cada sección de este capítulo, se podría escribir un capítulo completo, los conceptos aquí desarrollados son tan sólo generales, de forma que el lector se haga una idea de qué es y cómo funciona una computadora y se aleje de la idea de que la computadora posee “un duende mágico"que hace que las cosas pasen.

2.1. Máquinas de estados finitos

Primeramente es importante comprender un concepto matemático sobre máquinas y estados. Tomemos por ejemplo unas tijeras, en principio, son una máquina que posee dos estados, abierto y cerrado. Pero también posee una transi-

ción[1] entre los estados, es decir, una operación que permite hacer el cambio de estado, de abierto a cerrado y viceversa.

Por otra parte podemos analizar un semáforo como una máquina de estados, sin embargo, es también una máquina digital. Los estados de un semáforo son:

1. Luz Roja
2. Luz Verde
3. Luz Amarilla

La transición entre los estados se logra mediante un oscilador o temporizador que cada cierto tiempo, le indica al semáforo que debe cambiar su estado y pasar al siguiente estado. En este caso sólo hay una operación, que se moverse a la luz siguiente.

Una computadora también es una máquina de estados digital, pues lo único que hace es cambiar de un estado a otro, sin embargo el número de estados son millones y la forma de hacer cambios entre ellos es mediante operaciones matemáticas que se realizan en el procesador de una computadora.

Por comodidad, los posibles estados de una computadora, dado que escapan a la capacidad humana de imaginarlos completamente y por ende las transiciones se vuelven mucho más complicadas, se analizan como representaciones numéricas.

2.2. Sistemas numéricos

Los sistemas numéricos han sido estudiados por muchas generaciones. Normalmente el sistema utilizado actualmente es el sistema decimal y es denominado así porque está en base 10, lo que también quiere decir que se tienen diez símbolos diferentes para representar números, en este caso se tratan de los dígitos {0,1,2,3,4,5,6,7,8,9}.

Para comprender mejor los números en base 10, se muestra el siguiente ejemplo, suponga que tiene el número 193507.

Ese número en realidad es la suma de diferentes valores por potencias de diez:

```
1  9  3  5  0  7
|  |  |  |  |  |->  7x10^0
|  |  |  |  |---> + 0x10^1
|  |  |  |------> + 5x10^2
|  |  |---------> + 3x10^3
|  |------------> + 9x10^4
|---------------> + 1x10^5
```

Es decir, un número en decimal (y en cualquier base), es en realidad la suma de los dígitos multiplicados primero por la potencia de diez que corresponda

[1] Una transición quiere decir la forma en como se pasa de un estado a otro

según su posición relativa, iniciando desde cero con el último dígito a la derecha. Entonces 193507 es en realidad $1\times10^5+9\times10^4+3\times10^3+5\times10^2+0\times10^1+7\times10^0$. Una vez más, esto porque existe ya una convención sobre los diez dígitos que se van a utilizar en esta base y cada uno de ellos tiene un valor establecido, el cual es enseñado desde niños.

Sin embargo, el sistema decimal no es el único que se ha utilizado, los mayas utilizaban un sistema base 20, cuyos símbolos de los dígitos eran rayas y puntos (una raya implicaba cinco puntos) o los romanos utilizaban la base 4 (que es un poco diferente de la modificación que se utiliza actualmente), incluso los babilónicos utilizaban base 60[2].

2.2.1. Sistema binario y campos en una computadora digital

Dado que una computadora digital está compuesta por circuitos, los cuales trasladan electrones de un lado a otro, lo que se puede medir es la presencia de electrones o la ausencia de ellos, lo que permite asociar la presencia de electrones con un valor y su ausencia con otro valor. Tradicionalmente, si hay una diferencia de voltaje se toma como un 1, si no hay diferencia de voltaje (voltaje cercano a 0) se toma como un 0. Cada uno de estos valores son interpretados como dígitos de un número y son denominados *bits*.

Por otra parte, debido a que sólo tenemos dos estados posibles, esto nos obliga en cierta manera a utilizar un sistema binario.

Para facilitar la lectura y trabajo con los bits, se han hecho agrupaciones de bits los cuales son denominados diferente, en la siguiente tabla se muestra la forma en como se han denominado los diferentes grupos de bits.

Nombre	**Sigla**	**Tamaño**
Byte	B	8 bits
Kilobyte	KB	1024 Bytes
Megabyte	MB	1024 KB
Gigabyte	GB	1024 MB
Terabyte	TB	1024 GB
Petabyte	PB	1024 TB
Exabyte	EB	1024 PB
Zettabyte	ZB	1024 EB

Interpretación del sistema binario

Al igual que en los sistemas numéricos en el sistema binario los números son en realidad sumas de los dígitos, sólo que multiplicados por potencias de dos(recordemos que en decimal son potencias de 10).

[2]Los babilónicos fueron quienes midieron por primera vez el tiempo en segundos, minutos y horas, de ellos heredamos que un minuto sean 60 segundos y que 1 hora sean 60 minutos, porque era muy cómodo de representar en su sistema numérico

Ejemplo:111010^3 entonces, tenemos que el número que está representado es de la forma:

```
1  1  1  0  1  0
|  |  |  |  |  |->  0x2^0  = 0
|  |  |  |  |---> + 1x2^1  = 2
|  |  |  |------> + 0x2^2  = 0
|  |  |---------> + 1x2^3  = 8
|  |------------> + 1x2^4  = 16
|---------------> + 1x2^5  = 32
```

Por lo tanto el número 111010 es $0 + 2 + 0 + 8 + 16 + 32 = 58$ en decimal.

Está demostrado matemáticamente que cualquier número racional se puede representar como la suma de potencias de un número entero cualquiera.

2.3. Las partes de la computadora: Arquitectura von Neumann

La historia cuenta que la primera computadora "oficial"(ENIAC) tenía diez bombillos, uno para cada posible número (porque en 1945 lo único para lo que se quería una computadora era para computar, es decir calcular valores más rápido de lo que lo haría un ser humano), cuando el gran matemático e investigador Jhon von Neumman lo vio escribió un documento criticando el trabajo de los ingenieros que hicieron ENIAC, pues cada serie de diez bombillos sólo estaban utilizando diez posibles estados (uno por cada bombillo) mientras que en realidad habían 1024 $(2^{10})^4$ estados posibles diferentes y desde ahí dedicó parte de su investigación a la computación y sus ideas iniciales han sufrido pocas modificaciones desde entonces.

Según von Neumann la computadora tenía que estar dividida en 3 partes principales:

1. **CPU**: En él se realizan los cálculos matemáticos o transiciones de estado de la máquina que es la computadora, ahí es donde realmente ocurre "la magia"de la computadora.

2. **Memoria**: En la memoria se encuentran los datos que van a ser operados en el procesador, esta división permite que el procesador no tenga que ser muy grande. Dado que el procesador puede disminuir su tamaño, gracias a la memoria, también disminuye el costo total de la computadora.

3. **Dispositivo de almacenamiento masivo**: Tradicionalmente se usaban cintas magnéticas como dispositivos de almacenamiento. La idea de esta parte consiste en poder conservar los datos cuando la computadora esté

[3] Es muy importante recordar que únicamente se pueden usar dos dígitos

[4] Al igual que los circuitos, los bombillos tienen dos estados, si se tratara de dos bombillos, se tienen 4 estados, que son PP, PA, AP y AA (donde A es apagado y P prendido), con 10 bombillos, se tiene desde PPPPPPPPPP, PPPPPPPPPA ... hasta AAAAAAAAAA

apagada. Aunque tiende a ser la parte más lenta de una computadora, también es la menos costosa y la de mayor capacidad.

2.4. Circuitos y compuertas lógicas

Hasta el momento se ha presentado un esquema básico de lo que es una computadora, sin embargo, es importante que el lector comprenda que todas las operaciones que ocurren en el procesador se realizan por medio de circuitos y que la forma de hacer que un circuito haga una cosa u otra, es mediante las compuertas lógicas, las cuales permiten o inhiben el tránsito de los electrones haciendo que se representen valores diferentes de acuerdo con parámetros de entrada.

2.4.1. Compuertas lógicas

El tener sólo dos valores posibles en un circuito hizo que tuviera mucha relevancia los estudios de ciertos matemáticos ingleses en el Siglo XIX, dentro de los que resaltan algunos nombres como Venn y Morgan, quienes tomando estudios que databan desde Sócrates, formalizaron o facilitaron el estudio de la lógica.

Una compuerta lógica tiene un comportamiento equivalente a los operadores lógicos de proposiciones vistos en la sección 1.1.1 de la página 14. Dichos operadores son los siguientes:

1. And: El and recibe dos entradas, tiene como propósito lanzar un 1 únicamente cuando ambas entradas son 1, en caso contrario (si alguna entrada es un 0) lanza un 0.

2. Or: El or recibe dos entradas, tiene como propósito lanzar un 1 cuando alguna de las entradas es un 1, sólo en caso de que ambos sean 0, lanza un 0.

3. X-or: El x-or recibe dos entradas, tiene como propósito lanzar un 1 cuando las entradas son diferentes, es decir, 1 y 0 ó 0 y 1, en caso de que sean iguales (11 ó 00), lanza un cero.

4. Not: El not recibe una única entrada y tiene el propósito de invertirla, es decir, si recibe un 1, lanza un 0 y viceversa.

Mediante la combinación de estas compuertas, se pueden hacer todas las instrucciones que se tienen micro-programadas en una computadora. La forma en como se representan estas compuertas se muestran en la figura 2.1:

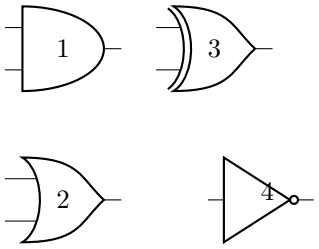


Figura 2.1: 1: AND, 2: OR, 3: X-OR, 4:NOT

2.4.2. Un sumador de dos bits

Para ilustrar un poco el ejemplo anterior, se va a desarrollar un gráfico de un sumador, que recibe dos bits y muestra la salida de lo que sería la suma de esos dos bits. Es importante recordar que cada bit es considerado como un número en binario.

Primero es importante recordar que tenemos una computadora y que es una máquina física por lo que los campos son limitados, fijos y que si un valor requiere menos bits (o sea, se puede representar con menos espacios de los predeterminados) se pueden rellenan esos bits con ceros a la izquierda dado que necesito dar un valor con una cantidad de bits fija.

Para comprender el sumador, veamos todas las posibles combinaciones en la siguiente tabla:

	Valor binario	Valor decimal
1+1	10	2
0+1	01	1
1+0	01	1
0+0	00	0

Se colocó adrede un cero a la izquierda en la columna “Valor binario"porque en una computadora siempre se tienen la misma cantidad de bits, aunque el número requiera menos dígitos, los restantes se rellenan con ceros, que al ser ceros a la izquierda, no alteran el valor. También es un tema interesante cuando el número requiere más de los bits que se tienen en el espacio asignado para el número, simplemente se toman más bytes y luego se procesa uno como la parte más a la izquierda y otro como la parte más a la derecha.

En la figura 2.2 se puede observar un circuito que hace exactamente esa misma función, se invita al lector que pruebe poniendo diferentes valores y confirmando que el resultado obtenido sea acorde con el mostrado en la tabla.

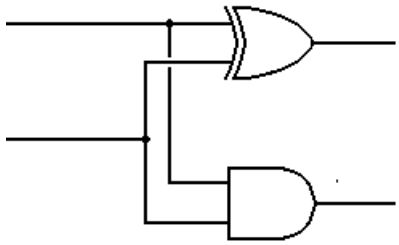


Figura 2.2: Sumador de dos bits

2.5. El procesador

El procesador o CPU, se compone de muchos de estos circuitos y no precisamente de dos bits. Actualmente[5] la más popular es de 64 bits, anteriormente la más popular era 32 bits. (Quizá sea interesante indicar que el primer procesador de 128 bits no fue para una computadora, sino para una consola de video juegos, el Playstation 2 de Sony y también resulta interesante que siempre sean potencias de dos[6]).

Entonces todos los circuitos de un procesador internamente son del ancho del procesador. Se supone que cuanto más sean la cantidad de bits que el procesador puede operar simultáneamente más datos va a poder trabajar y por ende más eficiente es su uso. Sin embargo, también cuanto más ancho sea el procesador, más complicados son los circuitos asociados.

En el procesador es donde ocurren todos los cambios de estado en una computadora. Por lo tanto, la función de transición entre estado está dada precisamente en los circuitos y los estados se almacenan en ciertos circuitos especializados en conservar su estado, denominados registros[7].

El procesador se divide en tres partes importantes, el decodificador(ver subsección: 2.5.1), la unidad de control(ver subsección: 2.5.2) y la ALU(ver subsección: 2.5.3).

2.5.1. Decodificador

La forma en como se indica cuál es el circuito o grupo de circuitos que deben utilizarse para hacer un cambio de estado es por medio de instrucciones, el conjunto de instrucciones son el alfabeto de la máquina (que representa cada una de las posibles transiciones que tiene) y a todas las secuencias posibles de estas instrucciones es denominado **lenguaje máquina**.

[5]Este libro fue escrito en el 2017

[6]Se recomienda al lector investigar la Ley de Moore

[7]Si el lector está interesado en conocer más sobre este tipo de registros se recomienda investigar sobre circuitos combinatorios y flip-flops o biestables

El decodificador es el circuito encargado de descubrir cuál es la instrucción siguiente por ejecutar en un proceso (un proceso es la secuencia de instrucciones).

Una vez que se sabe cuál es la instrucción siguiente, el decodificador "notifica.ª la unidad de control sobre dicha instrucción.

2.5.2. Unidad de control

La unidad de control es la encargada de activar o inhibir los circuitos correspondientes a la instrucción que fue decodificada anteriormente, también en caso de que la instrucción requiera parámetros, entonces esta es la encargada de recibirlos y pasarlos a la ALU.

2.5.3. ALU

La Unidad Lógico Aritmética (Arithmetic Logic Unit) por sus siglas en inglés, es la responsable de realizar todos los cálculos matemáticos en el procesador, como su nombre lo indica. Es aquí donde están los circuitos que realizan operaciones y cambian el valor de los registros.

2.5.4. Registros

Hasta el momento se ha mencionado brevemente qué son los registros, para aclarar cualquier duda al respecto, los registros son simplemente la memoria del procesador, es donde se almacenan los valores temporalmente durante las operaciones del procesador.

También los registros sirven para indicar errores en el cálculo, lo cual es algo muy común en los procesadores. Estos errores ocurren principalmente por una sobrecarga de electrones o por el contrario, por una falta de electrones. Para estos casos se cuenta con algunos registros que indican que no se debe traer la siguiente instrucción aún, sino que se debe repetir la intrucción anterior pues hubo un error.

Quizá una de las funciones más importantes de los registros, además de las mencionadas es el control que llevan sobre los procesos, es decir, se tiene un registro que indica el número de instrucción que se está ejecutando y se tiene también otro que indica cuál es la siguiente instrucción por ejecutarse.

2.5.5. Procesos

Anteriormente se ha mencionado que los procesos son secuencias de instrucciones, propiamente, estas secuencias de instrucciones representan un algoritmo (lo cual se profundizará más adelante en el libro).

Pero son los procesos, ese orden particular de instrucciones, lo que permite que la computadora haga las cosas que hace. Es muy importante resaltar que la computadora tiene un conjunto limitado de instrucciones, pero no por eso la cantidad de programas (los programas son en realidad conjuntos de procesos)

está acotada, es decir que es imposible que en algún momento se hagan todos los programas posibles que existen, porque ese conjunto es infinito.
¿Cómo es posible que con un conjunto finito de instrucciones se puedan hacer una cantidad infinita de procesos ***diferentes****?*
Para responder a esa pregunta, se recurrirá a esta metáfora, se sabe que el español tiene 27 letras, las cuales desde luego conforman un conjunto finito. Sin embargo, la cantidad de textos que se pueden escribir en español es infinita, cada obra literaria es única y si hubiera una obra máxima del español que tenga todas las posibles hileras que se pueden escribir en español, basta con agregarle al inicio "Esta es la hilera más grande del español" y ya se tiene una hilera diferente, aún más grande...
De la misma manera, aunque se tiene un conjunto finito de instrucciones, la computadora puede ejecutar un número infinito de procesos.

El compilador o el intérprete

Dado que las instrucciones son en realidad números binarios y resulta muy complicado para un ser humano escribir procesos o programas de esta forma, se han creado los lenguajes de programación.

Los lenguajes de programación son formas en que podemos escribir operaciones o funciones de manera "legible" para un ser humano, sin embargo, se necesita de un programa adicional que lo traduzca a lo que la computadora realmente entiende, este programa es el compilador o el intérprete.

El compilador genera un archivo ejecutable que es en realidad la secuencia de instrucciones en lenguaje máquina que se van a ejecutar. El intérprete por su parte, lo que hace es ir haciendo la traducción "en caliente" entre el código fuente del lenguaje de programación y las instrucciones que deben ejecutarse.

En otras palabras, se trata de un programa (es decir, una secuencia de instrucciones) que convierte un archivo de texto en una secuencia de nuevas instrucciones, que van a representar lo que estaba especificado en el archivo de texto, pero de manera diferente.

2.5.6. Pila (Stack)

Anteriormente se mencionó que la memoria del procesador son los registros, sin embargo, muchas veces los procesos deben utilizar más datos de los que tiene a mano en los registros, ya sea porque necesita los registros para otra cosa o porque tiene que llamar a un subproceso que se encarga de hacer otros cálculos y de devolver un valor.

Para estos casos, se cuenta con una sección de memoria denominada la pila, que consiste en un punto de memoria que se utiliza como auxiliar para guardar temporalmente *resultados parciales* de las operaciones que se van calculando en el procesador. Esta sección tiene un tamaño de muy pocos MB e incluso en ocasiones el espacio es menor al mega.

Aún así, este espacio es muy pequeño para poder guardar muchos datos, sólo permite guardar cosas pequeñas, no tan pequeñas como en el procesador, pero

no tan grandes como en la memoria.

2.6. Memoria

Cuando se menciona la memoria en general, se habla de la memoria RAM, las computadoras tienen actualmente unos 8 GB de RAM en promedio. Quizá el lector pueda preguntarse el por qué no se tiene una pila en la RAM y de esa manera nos ahorramos un poco el "salto"de un lado al otro, pero hay una condición importante, cuánto más grande es el tamaño en espacio de almacenamiento de bytes, también ocurre que es más lento.

Para hacer las cosas más eficientes, lo más importante, los cálculos actuales se conservan en niveles superiores de memoria y datos que tienen más tiempo de no utilizarse o que falta más tiempo para su utilización, se busca que se conserven en niveles inferiores. Por esta razón, donde masivamente almacenamos datos es en el disco duro.

2.7. Almacenamiento masivo: El disco

Los discos duros durante muchos años ha sido una secuencia de platos circulares de cintas magéticas puestos unos sobre otros, lo que permite que los datos se conserven aún cuando no perciba una corriente eléctrica (algo que no ocurre con ninguna otra parte de la computadora).

Es por este motivo que es ahí donde se almacenan archivos grandes, como películas, bases de datos, las aplicaciones (cuando se están ejecutando, se ponen en memoria), etc.

2.7.1. Otros mecanismos de almacenamiento masivo

En los últimos años han habido nuevas tendencias, como lo son las *stick memories* o los discos de estado sólido, que son híbridos entre sistemas de memoria con demanda de energía y los discos duros que no la necesitan del todo.

Pueden almacenar datos sin energía, pero para mostrarlos, sí la necesitan. La principal ventaja de ellos es que tienden a ser mucho más veloces que los discos duros, con la desventaja de que su precio es mucho mayor.

2.8. El bus de datos

Al inicio del capítulo, se mencionó que todos los cambios de estado en una computadora ocurren en el procesador, entonces debe existir un camino que comunique al procesador con todas las demás partes de la computadora. A ese camino se le conoce con el nombre de "bus de datos" y es el responsable de dos cosas, por un lado enviar datos al procesador y por otro traer datos del procesador (normalmente modificados o actualizados) para la memoria, el disco o cualquier otro componente.

2.8.1. Interrupciones

Finalmente una vez que se ha descrito de manera muy breve el funcionamiento de una computadora, hay un pequeño punto adicional que es importante destacar.

Una computadora ejecuta procesos, uno después de otro y estos procesos son secuencias de instrucciones. Entonces si una computadora tiene la instrucción de reproducir una canción, por ejemplo, cómo es que hace para hacer otras cosas "simultáneamente". Algunos pueden argumentar que las computadoras actuales tienen cuatro procesadores y por lo tanto pueden ejecutar cuatro cosas al mismo tiempo, pero de la misma manera, normalmente, vemos películas, bajamos cosas de internet, editamos documentos, movemos el mouse y abrimos nuevas ventanas, en fin, hacemos muchas más de cuatro cosas. Además, a finales del S. XX las computadoras tenían un solo procesador y aún así, se permitía hacer más de una cosa simultáneamente.

La forma en cómo se hace esta multiplexación[8] es mediante *interrupciones*.

De esta forma cada vez que movemos el mouse, llega un mensaje por la tarjeta de red (al usar un servicio de chat, por ejemplo), escribimos en el teclado, mandamos algo a imprimir o cualquier otro tipo de evento, se produce una interrupción. Dicha interrupción es colocada por el dispositivo que la provocó en el bus de datos, viaja hasta el procesador el cual interrumpe el proceso que se está ejecutando, guarda todos los valores de los registros en una sección especial de memoria denominada el TLB[9] (para poderlos recuperar después) y atiende la interrupción, que por lo general va a despertar a otro proceso.

De esta manera es que logramos hacer que la computadora ejecute varias acciones, lo que pasa es que lo hace tan rápido que nosotros no podemos darnos cuenta de los cambios por la rapidez con que éstos ocurren.

2.9. El sistema operativo

El sistema operativo es el componente de software que es el responsable de administrar qué proceso entra a correr cada vez que ocurre una interrupción, tiene la responsabilidad máxima de administrar apropiadamente todo, desde el procesador, pasando por la memoria, los dispositivos de almacenamiento masivo y hasta los dispositivos de entrada/salida o I/O[10]. Es así como un proceso que ha sido interrumpido, vuelve pronto al procesador.

Actualmente existen en el mercado muchos sistemas operativos, unos mejores que otros en su función de administrar los recursos de la computadora u orientados hacia alguna función particular, por ejemplo hay sistemas operativos que son muy buenos para que servidores web trabajen correctamente, pero que

[8] Aunque multiplexar no es una palabra aceptada en la Real Academia, se define en el ambiente técnico como el poder tener un mismo canal para diferentes tipos de señales, en este caso, procesos

[9] El Translation Look-aside Buffer es un componente especial de la memoria caché donde se guardan los estados de los procesos al ser interrumpidos

[10] Input/Output por sus siglas en inglés

serían muy malos en hacer películas animadas o viceversa. Quizá uno de los mejores técnicamente hablando para usuarios en general es GNU/Linux el cual es gratis y fácil de instalar en cualquier computadora.

2.10. Representación de datos

Para finalizar este capítulo, se mencionará brevemente como hace una computadora para representar datos, porque los usuarios son cada vez más demandantes y la capacidad de servicios ofrecidos es muy variada.

2.10.1. Números

Una de las primeras cosas que se representó en una computadora fueron números, los naturales son sencillos, porque tal y como se indicó al inicio del capítulo, en el fondo todo lo que una computadora maneja son números naturales (partiendo desde el cero).

Pero esto genera diferentes retos, como representar números negativos y números que llevan decimales.

Números negativos

En general la forma de poder representar números negativos es poniendo el bit más significativo como si fuera una potencia negativa de dos.

Asumamos que tenemos un número de un byte, es decir de ocho bits. Supongamos que se cuenta con este número: 10101011.

Si lo analizamos de la forma tradicional quedaría algo así:

```
1 0 1 0 1 0 1 1
| | | | | | | |->2^0 * 1 = 1
| | | | | | |--->2^1 * 1 = 2
| | | | | |----->2^2 * 0 = 0
| | | | |------->2^3 * 1 = 8
| | | |--------->2^4 * 0 = 0
| | |----------->2^5 * 1 = 32
| |------------->2^6 * 0 = 0
|--------------->2^7 * 1 = 128
                         +_____
                           171
```

Pero si tomamos el bit más significativo como una potencia negativa de dos, quedaría de la siguiente manera:

```
1 0 1 0 1 0 1 1
| | | | | | | |->2^0 * 1    =    1
| | | | | | |--->2^1 * 1    =    2
```

```
| | | | | |----->2^2 * 0    =    0
| | | | |------->2^3 * 1    =    8
| | | |--------->2^4 * 0    =    0
| | |----------->2^5 * 1    =   32
| |------------->2^6 * 0    =    0
|--------------->(-2)^7 * 1 = -128
                             +-----
                               -42
```

Y esta representación facilita mucho el poder realizar sumas y restas desde el punto de vista electrónico y que cumple con la posibilidad muy sencilla de facilitar el manejo de números enteros.

Números con decimales

También son llamados números con coma flotante o de punto flotante[11]. Es importante destacar que ese movimiento de la coma se puede en el mundo real, pero en el mundo computacional siempre se cuenta con una cantidad estática de bytes para poder representar todo lo que se requiere.

La solución es en realidad sencilla, hasta el momento se puede representar números enteros, entonces basta con poder indicar un número de coma flotante a partir de números enteros. Tomemos como caso 3,1415936, también lo puedo reescribir de la siguiente manera: 31415936×10^{-7} donde tanto el primer factor, como la base y el exponente del segundo factor, son números enteros.

Entonces para traducir eso a una computadora, basta con tomar una cantidad de bytes, de tamaño estático y poner una parte de estos bytes para un factor y otra parte de los bytes para el exponente del segundo factor (la base del segundo factor siempre va a ser dos, al ser binario).

2.10.2. Textos

En el fondo los textos son letras o caracteres, enlazados. La idea es muy sencilla, consiste en darle a cada caracter un número. Tradicionalmente se ha utilizado el valor ASCII, el cual utiliza un byte por cada letra, sin embargo, dado que los bytes sólo tienen 256 posibles valores (2^8) diferentes, esta tabla de valores dejaba por fuera caracteres en mandarín, árabe o japonés. De ahí que naciera UNICODE que asigna dos bytes a cada letra, el cual incluye todos los caracteres de los lenguajes citados y permite tener 65536 posibles valores (2^{16})

2.10.3. Imágenes

Cuando aparecieron los monitores a colores, las personas empezaron a querer ver archivos con imágenes, entre otras cosas. Los monitores a colores en realidad

[11] En particular porque la coma o el punto "flotan" en el número hacia la izquierda o la derecha de una posición particular y por el contrario, en los números enteros la posición es estática

se dividen e pixeles, cada pixel proyecta de alguna manera tres haces de luces, uno rojo, otro verde y uno azul (RGB por sus siglas en inglés).

Entonces una imagen, es una matriz de n pixeles de ancho por m pixeles de largo, cada pixel se representa con tres (en algunos caso 4) bytes, donde se indica la intensidad de rojo (red), verde (green) y azul (blue).

2.10.4. Música y sonido

La música y el sonido es en realidad una onda, en física la onda tiene algunos atributos, como frecuencia, amplitud y longitud de onda, todos son números y por lo tanto son representables en una computadora a partir de esas características.

2.10.5. Vídeos

Finalmente un vídeo es la unión de imágenes y sonidos, por lo que basta coordinar la forma en como se refresca una imagen con respecto a la frecuencia de sonido.

2.11. Problemas

2.11.1. Máquinas de estados

Describa los estados y las transiciones de las siguientes máquinas:

1. Una máquina contestadora
2. Una máquina dispensadora de refrescos
3. Una alarma de incendios

2.11.2. Conversión de bases

A continuación se muestra una tabla, donde las columnas son las bases deseadas, las filas son los diversos números. Complete la tabla, la primera fila está hecha a modo de ejemplo.

10	**2**	**3**	**5**	**7**	**8**
21	*10101*	*210*	*41*	*30*	*25*
31					
					46
	101110				
				1000	
				1110	
	1000				
		1120			
			44101		
59					

2.11.3. Sobre sistemas operativos

Investigue al menos 4 sistemas operativos diferentes, realice un cuadro comparativo con la información investigada donde se establezcan diferencias en cuanto a precio, facilidad de uso, popular entre los programadores[12], si posee virus que lo puedan atacar, en caso de que sea positivo, precio de los antivirus y tamaño en GB.

2.12. Resumen

En este capítulo 2 se explica brevemente el concepto de lo que es una computadora. Partiendo de los conceptos de máquinas de estados y pasando por los sistemas numéricos, los cuales son importantes para traducir los conceptos que manejamos a un formato que la computadora pueda procesar.

Luego se mencionaron los componentes básicos de la computadora como el procesador, la memoria y el disco duro con una respectiva descripción de su funcionamiento. El sistema operativo es el primer componente de software que se menciona en este libro, el cual es el encargado de administrar todas las partes de la computadora, a saber: procesador, memoria, disco duro y entrada/salida.

Finalmente se explica la técnica para poder representar los datos en una computadora, que se trata de la utilización de teoría de números para codificar la información presente en cada tipo de archivo.

En el capítulo 3 se iniciará con el concepto de algoritmo, el cual es lo que permite que la computadora, a través del procesador, pueda cambiar entre los posibles diferentes estados de la máquina. Sin embargo, el algoritmo per sé, nace como un concepto abstracto miles de años antes que la computadora, por lo que se iniciará con los conceptos asociados a algoritmos y funciones.

[12]Para esto no basta con preguntarle a los cuatro vecinos que son programadores cuál sistema usan

Capítulo 3

Algoritmos, fórmulas y funciones

> Si la gente no piensa que las matemáticas son simples, es sólo porque no se dan cuenta de lo complicada que es la vida
>
> John von Neumann

> Es altamente emocionante poder ver ejecutando algoritmos concebidos por uno mismo en una computadora
>
> Georges Alfaro

En programación, todos los conceptos y trabajos que se desarrollan tienen un fuerte transfondo matemático. En particular, el tema de algoritmos que viene de una palabra árabe y que es una secuencia ordenada de pasos para ejecutar algún proceso. Este capítulo es introductorio a algunos de esos conceptos.

3.1. Lenguajes

Un lenguaje es un conjunto de hileras que cumplen con alguna propiedad. Esta propiedad puede ser **formal** o puede ser **natural**, lo que los diferencia es que, en primera instancia, puede ser reconocida por una máquina de turing, mientras que la segunda, aún no sabemos si se puede procesar.

3.1.1. Lenguaje natural

El lenguaje natural es aquel que usamos diariamente, está asociado a una semántica y los seres humanos somos capaces de entenderlo aunque no esté bien elaborado: por ejemplo, la frase *Yo hablar español* no es una hilera que respete las reglas del español, sin embargo, es comprensible. Por el contrario la hilera *Estamos lloviendo bonito* sí es una hilera que está bien construida, aunque carece de sentido.

3.1.2. Lenguaje formal

Los lenguajes formales, por su parte son aquellos que son reconocidos por máquinas de estados como las computadoras. De ahí que los lenguajes de programación, al ser lenguajes formales, son reconocidos e interpretados por las computadoras y sobre ellos es que se escribe el software que de alguna u otra forma están asociados a un algoritmo.

3.2. Algoritmo

El primer punto que debe ser comprendido por cualquier persona que está aprendiendo a programar es el concepto de algoritmo. Dado que es el pilar sobre el cual se realizan todos los procesos en una computadora, desde hacer cálculos sencillos como sumas, hasta mostrar y digitalizar películas con figuras tridimensionales y sonido estéreo.

Tal y como se vio en el capítulo 2 una computadora es una máquina que ejecuta instrucciones, las secuencias de instrucciones son algoritmos. Todo algoritmo tiene un trabajo asociado, por trabajo se quiere decir, número de pasos.

La forma en como se le indica a la computadora qué ejecutar, es usualmente mediante un *lenguaje de programación*. Actualmente hay cientos de lenguajes de programación y cada año nacen nuevos lenguajes, por lo que el problema de decidir el lenguaje apropiado para aprender a programar es un reto y en general depende mucho de las habilidades que se espera que el programador desarrolle.

Vale la pena destacar que independientemente del lenguaje que se escoja para hacer programas, lo que tienen en común es que siempre permiten representar algoritmos, los cuales son traducidos en instrucciones ejecutables por una computadora.

3.2.1. Definición de algoritmo

Un algoritmo se define como una secuencia de pasos ordenada, específica, normalmente utilizada para la solución de algún problema. Existen dos tipos de algoritmos, los determinísticos y los probabilísticos, los segundos están asociados a una distribución de probabilidad, mientras que los primeros para una entrada x siempre se obtiene una salida y.

3.2.2. Definición de algoritmos determinísticos

Un algoritmo determinístico consiste en una secuencia de pasos que al seguirse al pie de la letra, ***siempre*** se obtiene el mismo resultado.

Para ejemplificar lo que es un algoritmo determinístico, a continuación se muestran dos recetas de cocina:

Receta #1	Receta #2
Ingredientes:	Ingredientes
300 gramos de tomates	3 tomates
4 latas de atún	4 latas de atún
Instrucciones	Instrucciones
Maje los tomates	Maje los tomates
mezcle con el atún y	mezcle con el atún y
agregue 0,1 gramos de sal	agregue sal al gusto

Como se puede ver la receta #1, se puede reproducir completamente de manera exacta en todos sus pasos y si se repite n número de veces, entonces se obtendrá en principio, el mismo resultado. Sin embargo, la segunda receta, no posee esta característica, por ejemplo, tiene una parte que indica *sal al gusto* y dado que cada persona tiene un gusto diferente para la sal, eso hace que la reproducción de los pasos no sea siempre exacta, por lo que la receta #2 **no** es un algoritmo en el sentido formal, mientras que la receta #1 **sí** se trata de un algoritmo.

3.2.3. Definición de algoritmo probabilístico

Un algoritmo se considera probabilístico cuando una ejecución da como resultado un valor distinto a otra ejecución, sin embargo, al hacer un análisis de una población de resultados, es decir, al analizar una gran cantidad de datos de las ejecuciones del algoritmo, éste se comporta de una manera asociada a una distribución de probabilidad.[1]

Los algoritmos probabilísticos son menos comunes en ejercicios para aprender a programar que los algoritmos determinísiticos, sin embargo tienen una gran aplicación en la realización de simulaciones de computadora.
Un ejemplo de un algoritmo probabilístico es lanzar un dado. Se puede detallar la forma en cómo tomar el cubo del dado, puedo incluso detallar la fuerza con que se arroja el dado, sin embargo, no puedo tener control sobre el viento o la superficie de rodamiento. Por otra parte, sí puedo *determinar* la *probabilidad* de cada resultado. Así, se espera que la probabilidad de obtener cualquiera de las seis opciones es $\frac{1}{6}$. De manera que si repito la ejecución del algoritmo n veces, siendo n un número suficientemente grande, se esperaría un comportamiento similiar al que se muestra en la siguiente tabla:

[1] Una distribución de probabilidad es aquella donde a cada valor resultante posible se le asigna una posibilidad de ocurrencia, este valor está dado entre 0 y 1, donde cero quiere decir que nunca ocurre y 1 que siempre ocurre

Resultado obtenido	Frecuencia esperada
1	$\frac{n}{6}$
2	$\frac{n}{6}$
3	$\frac{n}{6}$
4	$\frac{n}{6}$
5	$\frac{n}{6}$
6	$\frac{n}{6}$

Nótese que cuanto mayor sea el n, más se aproximarán las frecuecias obtenidas tras la ejecución del algoritmo a las frecuencias esperadas. Es decir, si el dado se lanza 6 veces, habría algunos números que salen dos veces y otros que no salen, si por el contrario se lanza el dado 6000 veces, es más probable que hayan cerca de 1000 apariciones por número. Por otra parte, en este ejemplo, se sigue una distribución uniforme, sin embargo, existen otras distribuciones que tendrían una repercusión que mapeada a este ejemplo, sería como si se cargara el dado para que algún resultado tuviera mayor frecuencia que otro.

3.3. Funciones

Una de las formas más básicas de un algoritmo es una función. Tal y como se describe en el capítulo 1 de la página 13 Se trata de la relación entre dos conjuntos. El conjunto de partida o dominio, donde tomamos las x y el conjunto de llegada o ámbito, donde tomamos las y.

Se escriben de manera genérica $f(x) = y$, a la variable x, normalmente se le llama parámetro de la función o *preimagen* y a la variable y se le denomina resultado o *imagen*

Todas las funciones que se muestran a continuación, son basadas en algoritmos determinísticos.

3.3.1. Ejemplos

A continuación se muestran algunas fuciones y se explican la secuencia de pasos (algoritmo) para poder obtener el resultado.

1. $f(x) = x$: Esta es la función identidad, que básicamente recibe un parámetro (x) y retorna el mismo valor que x posea.

2. $f(x) = x + 1$: Esta es la función sucesor, para un parámetro entero (x) retorna el entero siguiente.

3. $f(x) = x^2$: Esta es una función cuadrática. El algoritmo para obtener dicha función es un poco más largo que el ejercicio anterior. Se trata de tomar el valor de entrada y regresar la multiplicación por sí mismo, una vez.

4. $f(x) = 2x + 5$: Esta es una función lineal, donde la secuencia de pasos correcta es: primero multiplicar por dos el parámetro y a ese resultado sumar 5.

Es importante que se note que el resultado depende directamente del parámetro. Un parámetro diferente, provoca, en la mayoría de los casos, un resultado diferente. Pero este hecho, no afecta el determinismo de la función, pues para el mismo parámetro de entrada, siempre se produce la misma salida.

3.4. Función por partes

Una función que tiene al menos dos comportamientos diferentes con variables diferentes, se le conoce como función por partes. Normalmente el comportamiento de la función está dado por una condición lógica en alguno de sus parámetros. A esta condición lógica que permite diferentes comportamientos se le conoce con el nombre de *bifurcación*, porque *"bifurca"* el flujo de ejecución de un algoritmo.

3.4.1. Ejemplos

Hay muchos ejemplos de funciones por partes, a continuación se enlistan algunas:

1. $f(x) = |x|$
   La más común de las funciones por partes es la de valor absoluto. La condición es muy sencilla, si x es menor que cero, entonces la respuesta es $-x$, en caso contrario (es decir si es mayor o igual a cero), la respuesta es x. En general las funciones por partes se escriben de la siguiente manera:

$$f(x) = \begin{cases} -x & x < 0 \\ x & x \geq 0 \end{cases}$$

$$raiz(x) = \begin{cases} \sqrt{x}i & x < 0 \\ \sqrt{x} & x \geq 0 \end{cases}$$

   Otra de las funciones por partes es la de raíz cuadrada, tradicionalmente se dice que la raíz cuadrada de números negativos no existe, sin embargo, hay un conjunto de números (fuera de los reales) cuyo nombre son los números complejos. Básicamente la raíz de un número es, la raíz si se trata de un número positivo y si se trata de un número negativo, al resultado de la raíz se le multiplica por i, donde se sabe que $i = \sqrt{-1}$

$$p(x) = \begin{cases} verdadero & x & es & par \\ falso & en & otro & caso \end{cases}$$

   Esta función retorna un valor de verdad (verdadero o falso), retorna verdadero si el valor ingresado es par y falso si no lo es. Tradicionalmente las funciones se les ingresan números y retornan números, pero en computación y muchas áreas de la matemática, se necesitan funciones que retornen otras condiciones. Como en el caso anterior, donde se retorna un valor

de verdad ante un número. Es importante destacar que una función puede retornar otros valores diferentes a los números, aún cuando reciba números. Este tipo de funciones pueden ser utilizadas como funciones auxiliares en otras funciones más complejas.

$$mitad\_entera(x) = \begin{cases} x/2 & si \quad p(x) \\ (x-1)/2 & en \quad otro \quad caso \end{cases}$$

Esta función devuelve la mitad entera de un número, es decir, la mitad sin decimales. Si el número es par, la mitad es muy sencilla de obtener, basta con dividir el parámetro x entre dos, pero si es un número impar, antes de dividir a x entre dos, primero se resta 1 (para obtener el número par anterior) y luego se divide entre dos. Tal y como se mencionó en el inciso anterior, este es un ejemplo donde se utiliza una función condicional para indicar el comportamiento de la función principal.

3.4.2. Divide y vencerás

Divide y vencerás es un principio muy popular en computación, decía Descartes: "*Trabajo pesado es normalmente la suma de trabajos sencillos de fácil resolución*", este principio se aplica mucho en programación, donde para resolver un problema que parece grande, simplemente se busca *dividir* el problema en funciones más sencillas, de modo que la apropiada ejecución de las diferentes partes, resuelve el todo completo.

Comprender apropiadamente esta técnica es fundamental para hacer programas legibles, fáciles de mantener y que permiten reutilizar funciones, dado que son tan pequeñas que siempre van a poder ser utilizadas en más de un problema.

3.5. Fórmulas

Una fórmula es en sí misma una función, con la particularidad de que puede recibir más de un parámetro. Desde el punto de vista meramente matemático, una función puede recibir n elementos, porque puede ver esos elementos como un solo elemento, como una tupla[2]

3.5.1. Algunas fórmulas clásicas

1. $V(S,T) = \frac{S}{T}$
   A esta función se le llama velocidad (V), los parámetros que recibe son distancia(S) y tiempo (T). El resultado, se obtiene al dividir distancia entre tiempo. Muchos de los problemas de física clásicos, son en principio funciones de dos variables.

[2]Una tupla es una secuencia de elementos de diversos conjuntos. Por ejemplo, los puntos en un plano cartesiano (1,4),(-3,6),(0,0), etc, son tuplas de dos elementos

2. $E(m, c) = mc^2$
   Una de las fórmulas más famosas ($E = mc^2$), expresada como una función, donde se tienen dos parámetros, la masa y la velocidad a la luz. De modo que el algoritmo es tomar m, multiplicarlo por c y tomar ese resultado y multiplicarlo por c una vez más.

3. $sc(\alpha, r) = \frac{\alpha * r^2 * \pi}{360}$
   En este otro caso tenemos el área del sector circular, una vez más expresado como una función, en términos del ángulo de abertura(α) y el radio(r).

3.5.2. Sucesiones

Otra forma tradicional de las fórmulas son las llamadas sucesiones, también conocidas por muchos como series. Veamos un ejemplo:
$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + ... + 100$

Una representación equivalente a dicha secuencia es: $\sum_{i=1}^{100} i$

La importancia de esta notación radica en su simplicidad para escribir secuencias largas de números (en este caso sumandos) de manera muy sencilla y abreviada.

Para ejemplificar mejor el uso de Σ veamos las siguientes equivalencias:

$2 + 4 + 6 + 8 + 10 + 12 + ... + 30000000 = \sum_{i=1}^{1500000} 2i$
$1 + 3 + 5 + 7 + 9 + 11 + ... + 1000001 = \sum_{i=0}^{500000} 2i + 1$
$1 + 4 + 9 + 16 + ... + 1000000 = \sum_{i=1}^{1000} i^2$
$1 + \frac{1}{3} + \frac{1}{9} + \frac{1}{27} + \frac{1}{81} + ... + \frac{1}{14348907} = \sum_{i=0}^{15} \frac{1}{3^i}$
$12 + 15 + 18 + ... + 9999 = \sum_{i=4}^{3333} 3i$

El comportamiento general que tiene la notación Σ es:

$$\sum_{i=k}^{n} f(i) = f(k) + f(k+1) + f(k+2) + f(k+3) + ... + f(n)$$

Siempre se debe cumplir $n \geq k$

Como se puede apreciar Σ nos puede abreviar mucho tiempo para escribir sumas. Para el caso de multiplicaciones se utiliza Π, pero la secuencia es la misma, sólo cambia el operador, por ejemplo:

$$1 \times 2 \times 3 \times ... \times 100 = \prod_{i=1}^{100} i$$

3.6. Notación algorítmica

Como se indicó en la primera parte, un algoritmo es una secuencia de pasos y una computadora es capaz únicamente de ejecutar esta secuencia de pasos, con la particularidad de que ejecuta los pasos sin fallar y a una velocidad mucho mayor que la que sería capaz de ejecutar un ser humano.

Es por este motivo que es tan importante poder escribir apropiadamente esa secuencia de pasos de modo que la computadora lo pueda seguir.

Si bien es cierto, cada lenguaje tiene su propia forma de escribir un algoritmo [3] en este capítulo vamos a enfocarnos en escribir el algoritmo de la forma más estándar que sería pseudocódigo de C y funciones.

3.6.1. Notación general

En la siguiente sección se muestran ejemplos de como se puede expresar en términos de pasos una serie de fómurlas varias.

Duplo

Este algoritmo se trata de algo sumamente sencillo, consiste en dado un número, hallar el doble de dicho número, independientemente de si se trata de un número entero o racional, lo mismo que positivo o negativo.

Fórmula

$2 \times n$

Algoritmo

```
duplo(parámetro n)
    retornar 2*n
```

Fórmula general X1

Este algoritmo consiste en encontrar el primer cero de una ecuación de grado dos. (Como ejercicio propuesto, queda el encontrar el segundo cero)

[3] Ver anexo B del libro: Generalidades de lenguajes de programación

Fórmula
$x1 = \frac{-B+\sqrt{B^2-4AC}}{2A}$
Algoritmo

```
form_general_x1(parámetros A, B, C)
      discriminante <- B*B-4*A*C
      raiz_disc <- sqrt(discriminante)
      retornar (-1*B + raiz_disc) / (2 * A)
```

Notemos ciertas características en este algoritmo, como lo son los términos *discriminante* y *raiz_disc*. A estos valores se le conocen con el nombre de ***variables*** y se utilizan principalmente como valores auxiliares que facilitan (y en algunos casos posibilitan) la ejecución del algoritmo. También tienen la característica de que en muchos casos se cuenta con variables para hacer más fácil la lectura del código, por lo que es muy recomendable utilizar nombres de variables significativos.

3.6.2. Cálculo de precio con impuestos

Este es un problema clásico ya que en muchos lugares no se pone el precio con impuestos, dado que se cobra diferente tasa de impuestos en diferentes estados. Los parámetros necesarios son entonces, el monto antes de impuestos y el porcentaje de impuesto.
Fórmula $precioFinal = precio + precio * porcentaje$

Algoritmo

```
precio_final (parámetros precio, impuesto)
     precio_parcial <- precio*impuesto/100
     retornar precio+precio_parcial
```

3.6.3. Valor absoluto

Este se trata de un algoritmo conocido y mencionado anteriormente, consiste en obtener el valor absoluto de un número x.
Fórmula $|x|$
Algoritmo

```
valor_absoluto (parámetros x)
     si (x<0)
          retornar -1*x
     en otro caso
          retornar x
```

Tal y como se había mencionado, este algoritmo tiene una bifurcación basada en una condición lógica, en este caso la condición de si se trata de un número negativo o no.

3.6.4. Mayor

Este se trata de un algoritmo sencillo, recibe dos números y devuelve el mayor de ellos.
Fórmula $max(x, y)$
Algoritmo

```
mayor(parámetros x,y)
     si (x > y)
         retornar x
     en otro caso
         retornar y
```

Tal y como se había mencionado, este algoritmo tiene una bifurcación basada en una condición lógica, en este caso la condición de si se trata de un número negativo o no.

También existe una fórmula directa que debiera ser sencilla de comprender:

```
mayor(parámetros x,y)
     retornar (x+y+valor_absoluto(x-y))/2
```

La idea es sencilla, si sumo los dos elementos más la diferencia de ambos, entonces se tiene dos veces el máximo. Se invita al lector a demostrar el porqué esa fórmula funciona. La ventaja de esta función es que se ahorra el uso del comparador.

3.6.5. Suma de consecutivos

El siguiente presenta una nueva característica:
Fórmula $\sum_{i=1}^{n} i$
Algoritmo

```
1   suma(parámetros n)
2       total <- 0
3       i <- 1
4       mientras (i <= n)
5            total <- total+i
6            i<-i+1
7       retornar total
```

Este algoritmo tiene una condición, pero a diferencia de los casos anteriores donde la bifurcación era lineal, es decir, si pasa una cosa, ocurre A y si pasa otra cosa, ocurre B y listo, en este algoritmo las acciones se *repiten* hasta que la condición deje de ser verdadera (se espera que las acciones que se repiten modifiquen de alguna manera la condición para que en algún momento deje de ser verdadera).

A esta acción de repetición de acciones se le conoce con el nombre de ***ciclo***. En este caso particular, el ciclo está dado en las líneas 5 y 6 y la condición en

la línea 4. Al hacer un análisis línea por línea notamos que existen dos variables importantes, la primera es el total (línea 2), que viene a ser donde voy a estar guardando la respuesta temporal de las sumas parciales, hasta llegar al gran total. El otro es el i (línea 3), el cual viene a indicar el número de sumando que se está calculando.

Para explicar mejor el ejemplo, veamos como las variables cambian en cada *iteración* [4]

Supongamos un $n = 5$, en ese caso estaríamos calculando $total = 1+2+3+4+5$

Comencemos con el algoritmo:

Variable	Valor inicial	it01	it02	it03	it04	it05
total	0	1	3	6	10	15
i	1	2	3	4	5	6
	¿1 <= 5?	¿2 <= 5?	¿3 <= 5?	¿4 <= 5?	¿5 <= 5?	¿6 <= 5?

Se puede apreciar que las iteraciones se repiten (a total se le suma i y a i se le suma 1) en tanto la condición siga siendo verdadera, cuando deja de serlo, en este ejemplo, cuando i con un valor de 6, deja de ser menor o igual que cinco, entonces se rompe el ciclo y se pasa a la siguiente instrucción, luego del ciclo.

3.6.6. Suma de cuadrados

Consideremos una pequeña modificación al problema anterior la cual consiste en que, en lugar de sumar los primeros n números naturales, vamos a sumar los primeros n números naturales, pero elevados al cuadrado.

Fórmula $\sum_{i=1}^{n} i^2$

Algoritmo

```
1   suma(parámetros n)
2       total <- 0
3       i <- 1
4       mientras (i <= n)
5            total <- total+i*i
6            i<-i+1
7       retornar total
```

Como vemos, el algoritmo es casi igual, con la salvedad de que en lugar de sumar a total el valor de i, sumamos su cuadrado.

3.7. Problemas

A continuación se muestra una serie de problemas, se recomienda que su resolución sea en el mismo orden en que están planteados.

[4] Una iteración es el nombre que lleva el conjunto de acciones que se realizan en cada llamada de un ciclo

3.7.1. Algoritmos

La primera parte de esta sección de ejercicios consiste en un enfoque teórico de los algoritmos

Definición y reconocimiento de algoritmos

1. Indique cinco actividades de la vida contidiana que siguen un proceso determinístico. Indique si requieren de parámetros y en qué consiste.

2. Indique cinco algoritmos probabilísticos, indique en qué consiste la distribución de probabilidad de cada uno. Explique qué parámetros reciben según sea el caso.

Reconocimiento de algoritmos

Para cada uno de los siguientes ejemplos, explique si se trata de un algoritmo o no (es decir, si se puede hacer un algoritmo que cumpla dicha función). En caso de ser un algoritmo, explique si se trata de algoritmo determinístico o probabilístico.

1. Hablar en un lenguaje natural, como el español

2. Realizar un control de inventario

3. Jugar ajedrez

4. Componer una sinfonía

5. Cepillarse los dientes

6. Hacer una copia de un cuadro

7. Pintar un cuadro

8. Escribir un libro

9. Escribir un programa de computación (desarrollar el algoritmo)

3.7.2. Operaciones básicas

Para darle familiaridad con el lenguaje de programación seleccionado, programe cada uno de los siguientes ejercicios.

1. Sucesor: Consiste en recibir un número entero y devolver el siguiente a la derecha en la recta numérica.

2. Antecesor: Consiste en recibir un número entero y devolver el anterior a la izquierda en la recta numérica

3. Cuadrado: Consiste en recibir un número cualquiera y elevarlo al cuadrado

4. Cero X2: Consiste en recibir tres valores reales: A,B y C, que representan los coeficientes de los monomios que componen una ecuación cuadrática y devolver el segundo cero de la ecuación, es decir: $x2 = \frac{-B-\sqrt{B^2-4AC}}{2A}$

5. Mayor: Consiste en recibir 2 números e indicar cuál es el mayor de los dos

6. Menor: Consiste en recibir 2 números e indicar cuál es el menor de los dos

7. Hipotenusa: Consiste en recibir el largo de dos catetos y calcular, con base en ellos, el largo de la hipotenusa. Nota, se sabe que: $h^2 = c1^2 + c2^2$ donde h es hipotenusa y c1 y c2 son los catetos 1 y 2 respectivamente.

8. Desigualdad triangular: Consiste en recibir 3 números, e indicar con un valor booleano [5] si esos tres números cumplen con la desigualdad triangular, es decir, si la suma de los dos menores es mayor que el mayor de los lados.

9. Desigualdad triangular 2: Consiste en recibir 3 puntos (pares en un eje de coordenadas) e indica si éstos pueden ser los vértices de un triángulo.

3.7.3. Ciclos

Esta sección está destinada a que el estudiante razone algoritmos utilizando ciclos e iteraciones. Es importante destacar que no todos los lenguajes tienen esta característica, sino que hay algunos basados principalmente en la recursión, pero es un tema del siguiente capítulo, aún así es importante que el estudiante desarrolle el concepto independientemente de la implementación. Es importante señalar que no todos los ejercicios propuestos tienen una única solución.

1. MCD: Este problema consiste en recibir dos números enteros e indicar cuál es el máximo común divisor de ambos números.

2. mcm: Este problema consiste en recibir dos números enteros e indicar cuál es el mínimo común múltiplo de ambos números.

3. Suma de cubos: Para este ejercicio programe la siguiente fórmula (el n es recibido como parámetro) $\sum_{i=1}^{n} i^3$

4. Es primo?: Este problema consiste en recibir un número natural e indicar por medio de un *true* o un *false* si el número es primo o no (Verdadero si es primo y falso si no lo es). Un número primo es aquel que sólo posee dos divisores enteros, 1 y sí mismo.

5. n-simo primo: Este problema, similar al anterior, recibe un n y retorna el n-símo número primo, por ejemplo, si recibe un 1 devuelve un 2, porque 2 es el primer número primo. Si recibe un 4, entonces devuelve 7, porque es el cuarto número primo.

[5] Un valor booleano es un valor de sí o no, tradicionalmente representado por *true* y *false*

6. Cantidad de dígitos: Este programa busca crear un algoritmo que reciba un número entero con una cantidad de dígitos arbitraria y retorne la cantidad de dígitos que tiene ese número. Por ejemplo, si recibe 23544, devuelve 5.

7. Invertir un número: Este problema consiste en invertir el orden en que aparecen los dígitos de un número natural. Por ejemplo, si entra un 1853, entonces se retorna un 3581. Dentro de los casos curiosos, es que si entra el 1000, entonces tiene que retornar 1. (Nótese que 0001 tiene tres ceros a la izquierda y por lo tanto no cuentan)

8. Factorial: Este programa debe recibir un número n y calcular $n!$ el cual es el producto desde 1 hasta n. Por ejemplo: $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.

9. Suma de fracciones: Este problema consiste en programar la siguiente fórmula:
$\sum_{i=1}^{n} \frac{-1^i}{i}$
El n lo recibe como parámetro.

10. Suma de suma: Este ejercicio consiste en programar la siguiente fórmula:
$\sum_{i=1}^{n} \sum_{j=1}^{i} i * j$
Se recibe el n como parámetro

11. Producto: Este ejercicio consiste en programar la siguiente fórmula: $\prod_{i=0}^{n} \frac{1}{2^i}$

12. Producto de sumas: Este ejercicio consiste en programar la siguiente fórmula: $\prod_{i=2}^{n} \sum_{j=1}^{i^2} \frac{2i}{j}$

13. Números perfectos: En el 600 a.C. Pitágoras definió que un número era perfecto si la suma de todos sus factores (exceptuando sí mismo) era igual a ese número. Por ejemplo el 6 es el primer número perfecto, sus factores son 1,2,3 y $1 + 2 + 3 = 6$. Escriba un programa que reciba un número y retorne verdadero si es perfecto y falso si no lo es.

14. Números amigos: Pitágoras también trabajó con lo que son números amigos, un número a es amigo de un número b si los factores de a suman b y los factores de b suman a

    : Escriba un programa que reciba dos números y devuelva verdadero si son amigos y falso si no lo son.

    : Escriba un programa que reciba un número n y devuelva el número amigo de n, si lo tiene o retorne -1 si no lo tiene.

15. Según indica la conjetura de Goldbach para un número par, existen dos números primos tales que sumados den ese número. El ejercicio consiste en realizar un método que dado un número par que se recibe como entrada,

se retorna una lista de dos números primos que sumados dan el número par de la entrada.

16. Según el teorema de los cuatro cuadrados, existen para todo n, cuatro números enteros que elevados al cuadrado suman n. Su tarea consiste en crear un programa que dada una entrada n, imprima todos los posibles 4 números (a, b, c, d) tal que $n = a^2 + b^2 + c^2 + d^2$.

17. Según el teorema de Carmichael, para el n-simo ($n > 12$) número de fibonacci, existe un primo que es factor de ese número y que no es factor de ningún otro número de fibonacci previo. Realice un programa que para una entrada n, indique el número primo que divide al n-simo número de fibonacci. En caso de ser 1, 2, 6 o 12 (que son las únicas excepciones), devolver -1.

3.7.4. Conversor de bases

Realice una función que reciba tres parámetros, el primero es un número, el segundo es la base en que se encuentra ese número y el tercero la base a la que deseo convertirlo. Por ejemplo: $conversor(6, 10, 2) = 110$, porque está brindando el número 6 en decimal y desea pasarlo a base 2 o binario, la respuesta es 110 porque en binario representa el seis.

1. Realice este conversor de bases asumiendo que las posibles bases son del 2 al 10

2. Realice el conversor de bases para bases del 2 al 36, donde se usen letras para los símbolos superiores a 9. (Ejemplo: $A \equiv 10, B \equiv 11, F \equiv 15$ etc.)

3.8. Resumen

El este capítulo se definió el concepto de lenguaje y se diferenció sobre lo que es un lenguaje formal y un lenguaje natural. Sobre los lenguajes formales es que se pueden escribir los algoritmos, que son secuencias de pasos para realizar alguna acción.

También se diferenció entre los algoritmos determinísticos que son aquellos que siempre para una entrada, generan una misma salida y los probabilísticos que siguen una distribución de probabilidad.

Se indicó que una función es en sí misma, un algoritmo, la cual indica los pasos a seguir para uno o varios parámetros de entrada.

Finalmente se estableció un modelo de nomenclatura algorítmica para describir los algoritmos, similar a como algunos lenguajes de programación lo hacen.

En el capítulo 4 se verá el concepto de recursión, que se trata de una herramienta muy útil para escribir algoritmos y que se apoya en el uso de la pila del computador.

Capítulo 4

Recursión

> La iteración es humana, la recursión es divina
>
> L. Peter Deutsch

En términos de programación, la recursión es una de las herramientas que permite escribir código de forma más reducida (es decir, normalmente ocupa menos líneas) y a la vez permite escribir código con un alto nivel de estética y resulta muy elegante de leer. Sin embargo, es una herramienta que debe de ser utilizada con cuidado y como toda herramienta, es útil principalmente en manos de quien sepa utilizarla apropiadamente.

4.1. Funciones recursivas

La recursión se define como una llamada de una función a sí misma, pero en términos más simples, donde más simple implica más cercano a un caso base definido previamente. Toda función recursiva es una función por partes, donde hay al menos k casos base y la llamada recursiva.

4.1.1. Ejemplos de funciones recursivas

A continuación se definen algunas funciones recursivas para mostrar su simplicidad y para señalar en cada caso, las condiciones que tiene la recursión, es decir, caso base y caso recursivo.

Factorial

El primer ejemplo es factorial, en primera instancia porque omitirlo sería injustificable en una explicación de recursión. Factorial es una fórmula vista en el capítulo 3. Pero sin embargo se cumple una propiedad muy elegante:

$n! = n \times (n-1)!$ Lo que permite definir factorial en términos de sí mismo, es decir, reduzco el problema de calcular n a calcular primero $(n-1)!$ y luego tomar ese resultado y multiplicarlo por n. De igual manera, calcular $(n-1)!$ va a requerir primero calcular $(n-2)!$ para luego multiplicarlo por $(n-1)$ y así sucesivamente hasta, el caso base, que en este caso particular consiste en $0! = 1$.

$$factorial(n) = \begin{cases} 1 & si \quad n = 0 \\ n * factorial(n-1) & en \quad otro \quad caso \end{cases}$$

Como se aprecia en la función, se mantiene la peculiaridad que $n! = n \times (n-1)!$ y eso es precisamente lo que está indicado en la función.

También es importante mencionar que esta recursión es de grado 1, porque se cuenta con un único caso base $(n = 0)$, esto implica que la función llama al inmediato anterior $f(n-1)$, únicamente.

Para aclarar la forma de ejecución de esta función, veamos cómo es el proceso de cálculo para n=3. Se utiliza f en lugar de factorial, para acotar el desarrollo del ejemplo.

```
f(3) = 3*f(2)
         f(2) = 2*f(1)
                  f(1) = 1 * f(0)
                             f(0) = 1  **CASO BASE**
                  f(1) = 1 * 1   **Calculado f(0)
         f(2) = 2*1 **Calculado f(1)
f(3) = 3*2 **Calculado f(2)
6 **Calculado f(3)
```

Podemos apreciar cómo es que se realiza el cálculo de manera precisa, con la cualidad de que se tiene una recursión donde la forma de expresión algorítmicamente es muy sencilla, pero requiere de un doble proceso, uno de "ida"hasta el caso base y otro de "vuelta"hasta el valor inicial con el que fue invocada la función.

Suma de los primeros n naturales

En el capítulo anterior, se programó la fórmula: $\sum_{i=1}^{n} i$. Para efectos de este ejemplo, vamos a escribir una función recursiva que calcule el mismo valor.

$$suma(n) = \begin{cases} 1 & si \quad n = 1 \\ n + suma(n-1) & en \quad otro \quad caso \end{cases}$$

Una vez más, se aprecia cómo es que se puede sintetizar todo el proceso de cálculo, dado que la recursión se encarga de realizar la ejecución y rellenar los vacíos en el proceso. En esta ocasión, lo que se está utilizando es esta propiedad:

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i$$

Analicemos ahora la ejecución de esta función con un $n = 4$, se utilizará s en lugar de *suma* para acotar el desarrollo del ejemplo.

```
s(4) = 4+s(3)
          s(3) = 3+s(2)
                    s(2) = 2+s(1)
                              s(1)=1 **CASO BASE**
                    s(2) = 2+1   **Calculado s(1)
          s(3) = 3+3 **Calculado s(2)
s(4) = 4+6 **Calculado s(3)
10 **Calculado s(4)
```

Podemos apreciar como es que el caso base sigue siendo sólo uno y se mantiene el hecho de que se trata de una recursión de grado 1.

Fibonacci

Dentro de los grandes clásicos de las funciones recursivas está la función de fibonacci, se trata de una función de grado 2 y que genera una secuencia como se muestra en la siguiente tabla:

N-simo valor	1	2	3	4	5	6	7	8	9	10
Fibonacci(n)	1	1	2	3	5	8	13	21	34	55

Como se puede apreciar, el n-simo valor consiste en la suma de los dos valores anteriores, así por ejemplo, fibonacci(3) = fibonacci(2)+fibonacci(1), esta característica es lo que hace que sea de grado 2. Lo que nos lleva a la defínción formal de la función:

$$fibo(n) = \begin{cases} 1 & si \quad n = 1 \\ 1 & si \quad n = 2 \\ fibo(n-1) + fibo(n-2) & en \quad otro \quad caso \end{cases}$$

Veamos el comportamiento para $n = 5$

```
f(5)=f(4)+f(3)
     f(4) = f(3)+f(2)
            f(3) = f(2)+f(1)
                   f(2) = 1 **CASO BASE**
            f(3) = 1+f(1)
                     f(1) = 1 **CASO BASE**
            f(3) = 1+1
     f(4) = 2 + f(2)
                f(2) = 1 **CASO BASE**
     f(4) = 2 + 1
f(5)= 3  + f(3)
           f(3) = f(2)+f(1)
                  f(2) = 1 **CASO BASE**
```

```
          f(3) = 1+f(1)
                   f(1) = 1 **CASO BASE**
          f(3) = 1+1
f(5) = 3 + 2
5
```

Como se puede observar, f(3) fue calculado en dos ocasiones, una a raíz de f(5) = f(4)+**f(3)** y otra por f(4) = **f(3)**+f(2). Lo mismo ocurre, pero más veces, con f(2) y f(1).

La ejecución de esta función es mucho más compleja que las anteriores. Se trata de un caso donde incluso se repiten muchos valores calculados, cuanto mayor sea el n, mayor van a ser la cantidad de valores que se repiten en el cálculo. Al ser de grado dos, la cantidad de valores repetidos se acerca a 2^n.

Función de Ackerman

Ackerman es una de las funciones recursivas por excelencia. A continuación se puede ver la definición formal:

$$A(m,n) = \begin{cases} n+1 & si \quad m = 0 \\ A(m-1,1) & si \quad n = 0 \quad y \quad m > 0 \\ A(m-1, A(m,n-1)) & si \quad n > 0 \quad y \quad m > 0 \end{cases}$$

Esta es una muestra de como incluso, un parámetro de la función puede ser una invocación de la misma función recursiva, o incluso, puede ser otra función cualquiera. Otra diferencia que tiene esta función es el número de parámetros que recibe, pues como podemos ver, son dos parámetros.

Veamos la ejecución de la función con $m = 2$ y $n = 2$

```
a(2,2) = a(1,a(2,1))
         |   a(1, a(2,0))
         |   |    a(1,1)
         |   |    a(0, a(1,0))
         |   |    |    a(0,1)
         |   |    a(0,2)
         |   a(1,3)
         |   a(0, a(1,2))
         |   |    a(0, a(1,1))
         |   |    |    a(0, a(1,0))
         |   |    |    |    a(0,1)
         |   |    |    a(0,2)
         |   |    a(0,3)
         |   a(0,4)
         a(1,5)
         a(0,a(1,4))
         |   a(0, a(1,3))
         |   |    a(0, a(1,2))
```

```
        |   |    |    a(0, a(1,1))
        |   |    |    |    a(0, a(1,0))
        |   |    |    |    |    a(0,1)
        |   |    |    |    a(0,2)
        |   |    |    a(0,3)
        |   |    a(0,4)
        |   a(0,5)
        a(0,6)
a(2,2) = 7
```

4.1.2. La Pila

Tal y como se explica en el capítulo 2 la computadora trabaja con una pila, en el caso de la recursión, la pila se vuelve absolutamente impresindible porque es lo que permite "*recordar* por dónde va"la recursión y a través de ella es que se puede devolver, una vez calculados los elementos más internos de la recursión. Por ese motivo es que a la recursión sencilla, se le llama recursión de pila. La recursión de pila tiende a ser más ineficiente que las versiones iterativas del mismo programa.

4.2. Recursión de cola

La recursión de cola es la razón por la que la recursión es tan eficiente como la iteración. Se puede demostrar que para cualquier iteración, es posible realizar una función equivalente en complejidad y eficiencia con recursión de cola. Tal y como se explica, en la recursión previa (denominada de pila), la ejecución tiene una parte donde se llega al caso base y luego se devuelve hasta poder encontrar el caso inicial.
En la recursión de cola se busca evitar ese retroceso, de modo que se pasa como un parámetro el valor parcial que se está buscando.

4.2.1. Ejemplos de recursiones de cola

Es importante destacar que para hacer una recursión de cola, casi siempre es necesario invocar a una función auxiliar para poder agregar el parámetro que se está buscando. A continuación se describen varios ejemplos.

Factorial

Una vez más vamos a implementar factorial, pero esta vez utilizando recursión de cola. Primero veamos la versión matemática:

$$f(n) = f'(n, 1)$$

$$f'(n, acum) \begin{cases} acum & si \quad n = 0 \\ f'(n-1, n * acum) & en \quad otro \quad caso \end{cases}$$

Primero, se debe recordar que factorial es una función que sólo recibe un valor, por lo que para poder hacer apropiadamente la recursión de cola, se debe de utilizar una función auxiliar, en este caso llamada f'. A continuación se muestra la ejecución de esta versión para $n = 3$

```
f(3) = f'(3,1)
f'(3,1) = f'(2,3)
          f'(2,3) = f'(1,6)
                    f'(1,6) = f'(0,6)
                              f'(0,6) = 6
```

Como se puede apreciar, no hay necesidad de devolverse, y se mantiene el concepto de recursión, que es hacer una llamada a la misma función, pero en términos más simples.

Suma de los primeros n números naturales

Una vez más, utilizamos este ejemplo. Primero veremos la definición matemática.

$$s(n) = s'(n, 0)$$
$$s'(n, acum) \begin{cases} acum & si \quad n = 0 \\ s'(n-1, n+acum) & en \quad otro \quad caso \end{cases}$$

Veamos la ejecución para $n = 4$

```
s(4) = s'(4,0)
       s'(3,4)
       s'(2,7)
       s'(1,9)
       s'(0,10)
       10
```

Quizá lo interesante en la recursión de cola es que los valores se calculan de forma inversa de como se calculan utilizando recursión de pila.

Fibonacci

En el caso de fibonacci, la forma de lograr la recursión de cola involucra conceptos de programación dinámica, se invita al lector a que investigue por su cuenta un poco sobre estos conceptos. La idea general es buscar armar desde los casos base hacia delante la secuencia, en lugar de su versión tradicional (en recursión de pila), que parte del n-simo valor hacia atrás.
Primero se presenta la versión matemática:

$$f(n) = \begin{cases} 0 & si \quad n = 0 \\ f'(n,0,1,1) & en \quad otro \quad caso \end{cases}$$

$$f'(n,penu,ult,nsimo) \begin{cases} ult & si \quad n = \quad nsimo \\ f'(n,ult,penu+ult,nsimo+1) & en \quad otro \quad caso \end{cases}$$

Como ya es costumbre en este capítulo, veremos una ejecución de esta versión de fibonacci, para ello utilizaremos un $n = 3$

```
f(3) = f'(3,0,1,1)
       f'(3,1,1,2)
       f'(3,1,2,3)
       2
```

Una vez más podemos observar como es que no es necesario hacer el retorno de valores, sino que en una sola pasada, calculamos el valor de fibonacci de 3.

4.3. Problemas

A continuación, tenemos una serie de problemas que buscan ser introductorios a la recursión.

4.3.1. Operaciones básicas

Programe de forma de recursiva los siguientes ejercicios:
SUGERENCIA: primero escriba la función en forma matemática

1. mcm (a,b): Mínimo común múltiplo
2. MCD (a,b): Máximo común divisor
3. Es-Primo (n): Función que retorna verdadero si el n es primo y falso si no.
4. n-simo-primo(n): Función que retorna el n-simo número primo

4.3.2. Sucesiones

Programe primero con recursión de pila y luego con recursión de cola cada uno de las siguientes sucesiones:

$2+4+6+8+10+12+...+30000000=\sum_{i=1}^{1500000} 2i$
$1+3+5+7+9+11+...+1000001=\sum_{i=0}^{500000} 2i+1$
$1+4+9+16+...+1000000=\sum_{i=1}^{1000} i^2$
$1+\frac{1}{3}+\frac{1}{9}+\frac{1}{27}+\frac{1}{81}+...+\frac{1}{14348907}=\sum_{i=0}^{15}\frac{1}{3^i}$

4.3.3. Relaciones de recurrencia

Considere las siguientes series:

1. 1,1,1,3,5,9,17,31,57...
2. 3,6,12,24,48,96...
3. 1,3,7,17,41,99,239,577,1393...
4. 2,3,4,11,21,40,83,165,328...

Para cada una de las series anteriores, realice los siguientes ejercicios:

1. Escriba la función recursiva de pila de forma matemática, puede contar con que el primer elemento de cada serie es un caso base, pero no necesariamente el único, este número debe hallarlo, según considere conveniente para cada serie. Indique además el grado de la función.
2. Escriba la función recursiva de cola para las mismas series.
3. Programe las dos versiones de las funciones, tanto la recursión de cola como la recursión de pila.

4.3.4. Volviendo al pasado

Reescriba los programas realizados en la sección 3.7 de la página 59.

4.3.5. El algoritmo más antiguo que se conoce

El algoritmo de Euclides, es la forma más antigua para calcular el Máximo Común Divisor de dos números. Está basado en varias propiedades descritas en el capítulo uno de números enteros.

Suponga que se quiere encontrar el MCD de (a, b) para ello, necesitamos encontrar $d/d|a \wedge d|b$.

Sabemos que

1. $d|b \wedge d|a$ por hipótesis
2. $d|a \implies d|ak$ si d es factor de a, entonces es factor de los múltiplos de a
3. $b = ak + r$ Por el teorema de la división
4. $d|b - ak$ Si d divide a ambos, debe de dividir a la resta
5. $d|ak + r - ak$ Reemplazando (2) en (4)
6. $d|r$ En (5) realizando las operaciones respectivas

Por lo tanto todo divisor común de a y b también divide al residuo de ambos y como el máximo es uno de ellos, entonces también ocurre que divide al máximo de todos los divisores, éste tiene que ser el máximo de ellos.

Por lo tanto el algoritmo de euclides indica lo siguiente:

$$euclides(a, b) = \begin{cases} a & si \quad b = 0 \\ euclides(b, a(b)) & en \quad otro \quad caso \end{cases}$$

4.3.6. Torres de Hanoi

Investigue lo que son las Torres de Hanoi y escriba un programa recursivo que muestre cada uno de los pasos de los discos entre las tres diferentes torres.

4.4. Resumen

En este capítulo se definió el concepto de recursión, el cual está ligado a una función por partes, donde la principal característica consiste en casos base, junto con una llamada a la misma función en términos más simples, es decir, más cercanos al caso base.

La recursión se divide en recursión de pila y recursión de cola, dependiendo de la forma en como se hace la llamada recursiva. Se trata de una herramienta muy poderosa con la que se puede contar para realizar muchos ejercicios de programación en general. Por razones físicas, no siempre es posible completar en un tiempo razonable los problemas.

En el capítulo 5 se abordará el problema de la eficiencia, que ayudará a entender por qué unos programas duran más que otros y a calcular el tiempo aproximado que debe de ser consumido por un programa en su ejecución.

Capítulo 5

Eficiencia

> Soy vanidoso, pero no en lo que se refiere a mi apariencia y sí en cuanto a mi trabajo
>
> Harrison Ford

El poder programar correctamente tiene que ir muy ligado a hacerlo eficientemente. Se necesita mostrar claramente que los algoritmos no sólo cumplen la función para la que fueron creados, es decir, no basta con que se resuelva el problema, sino que se encuentre la forma más adecuada de resolverlo.

Un ejemplo para mostrar la importancia de la eficiencia, suponga que usted se encuentra en París y se le asigna llegar a Bélgica. Una manera es dirigirse hacia el norte, pasar la frontera y llegar a Bélgica. Otra manera es dirigirse al oeste, hacia Italia, pasar Torino y Milán, luego seguir al oeste hasta Eslovenia, en ese momento seguir al Norte, llegar a Austria. Una vez en Austria, se continua al norte y llegar hasta Alemania, pasando por Munich, subiendo hasta Berlín, una vez ahí, se dirige hacia Bremen, quedando cerca de la frontera con Holanda, llegando a este país para finalmente ir al sur y llegar a Bélgica.

La diferencia que se intenta mostrar, es que ambas rutas llegan a Bélgica, sin embargo, una es mucho más ***eficiente*** que la otra. En los algoritmos ocurre lo mismo, el número de pasos para poder alcanzar una solución es muy importante y es responsabilidad del programador el poder encontrar un algoritmo cuyo número de pasos sea mínimo para resolver el problema que está programando.

5.1. Time Complexity Function O grande

La *Time Complexity Function* o función O (se lee O *grande*) es una función que me indica el nivel de crecimiento en pasos que va a tener una función o algoritmo de acuerdo con el **tamaño** de los parámetros que recibe. Es muy

importante destacar que esta función nada tiene que ver con el tiempo de ejecución, sino únicamente con el número de pasos necesarios para completar el algoritmo en el **peor** caso posible. Resulta claro que a menor número de pasos, pues el tiempo necesario para poder ejecutar un algoritmo será menor, siempre y cuando se ejecuten en procesadores iguales con idénticos recursos de hardware. Es decir, el tiempo final que se necesita para ejecutar un paso, lo determina la computadora en que se va a ejecutar y qué tan rápido se ejecuten los pasos en ella.

Precisamente por este motivo es que el O debe ser completamente independiente del hardware, sino más bien debe de ser visto a la luz de un análisis matemático sobre el algoritmo que se está ejecutando.

En la figura 5.1 en la página 76 se puede apreciar diversos valores típicos para un O. Podemos ver como es que el número de pasos varía significativamente dependiendo de la función característica para un algoritmo.

Por ejemplo, si se tiene un algoritmo con $O(n)$ y recibe un parámetro de tamaño 5, entonces tardará 5 pasos en completarlo, pero si otro algoritmo tiene es $O(n!)$ y recibe un parámetro de tamaño 5, entonces tardará 120 pasos en completarlo.

Como podemos apreciar a raíz de esto, es importante buscar algoritmos con un O que pretenda ser el mínimo necesario para completar una tarea.

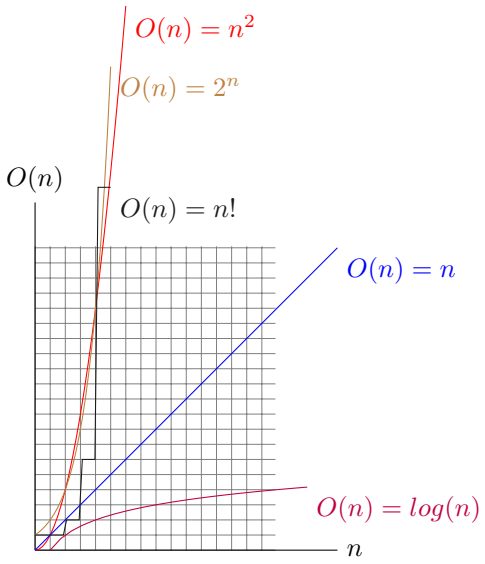


Figura 5.1: Diversos casos de complejidad

Para ejemplificar mejor la figura 5.1, se muestra una tabla con diferentes valores de n y la diferente cantidad de pasos que requieren para los diversos O y bajo el número de pasos se encuentra el tiempo que tardaría en una

computadora que fuera capaz de ejecutar 10 millones de instrucciones por segundo (aproximadamente 10 Ghz).[1]

n	$O(n) = log(n)$	$O(n) = n$	$O(n) = n^2$	$O(n) = n^3$	$O(n) = 2^n$	$O(n) = n!$
1	1 $1 \times 10^{-7} sec$	1 $1 \times 10^{-7} sec$	1 $1 \times 10^{-7} sec$	1 $1 \times 10^{-7} sec$	1 $1 \times 10^{-7} sec$	1 $1 \times 10^{-7} sec$
10	4 $4 \times 10^{-7} sec$	10 $1 \times 10^{-6} sec$	100 $1 \times 10^{-5} sec$	1000 $1 \times 10^{-4} sec$	1024 $1,024 \times 10^{-4} sec$	3628800 $0{,}362 sec$
25	5 $5 \times 10^{-7} sec$	25 $2{,}5 \times 10^{-6} sec$	625 $6{,}25 \times 10^{-5} sec$	15625 $15{,}625 \times 10^{-4} sec$	33554432 $3{,}35 sec$	$15{,}51 \times 10^2 4$ $4{,}91 \times 10^8 siglos$
50	6 $6 \times 10^{-7} sec$	50 $5 \times 10^{-6} sec$	2500 $25 \times 10^{-5} sec$	125000 $125 \times 10^{-4} sec$	$11{,}25 \times 10^{14}$ 3.5 años	$30{,}41 \times 10^{63}$ $96{,}44 \times 10^{46} siglos$
100	7 $6 \times 10^{-7} sec$	100 $1 \times 10^{-5} sec$	10000 $1 \times 10^{-3} sec$	10^6 $0{,}1 sec$	$12{,}67 \times 10^{27}$ $40{,}2 \times 10^{12} siglos$	$93{,}31 \times 10^{158}$ $29{,}5 \times 10^{140} siglos$

Quizá resulte interesante saber que la edad del universo se calcula, de acuerdo con la teoría del Big Bang, en 140 millones de siglos. Resulta curioso cómo es que algunos de los problemas descritos anteriormente duran mucho más que eso.

5.2. Problemas P y NP

Primero es importante destacar que P y NP son conjuntos de O. Los problemas P son aquellos cuyo O es **polinómicamente acotado** (de ahí el P, de polinomio), es decir, son la forma n^k donde n es el tamaño del problema y k es una *constante* cualquiera. Los problemas NP por su parte son aquellos cuyo O es **no polinómicamente acotado**, es decir de la forma k^n, incluso $n!$ y n^n pertenecen a esta familia. (A éstos dos últimos se les llama NP-Completos, NP-Duros o NP-Hard)

La razón de su importancia es que los problemas P tienden a crecer más lentamente en su número de pasos y por ende, en la ejecución de ellos. Esto facilita su ejecución y su cálculo. Tal y como podemos observar en la tabla de la página 77, donde las dos columnas de la izquierda crecen mucho más rápido en número de pasos y consecuentemente en tiempo de ejecución.
Incluso, su paralelización[2] tiene sentido, porque acota de alguna forma el tiempo de ejecución[3]. Sin embargo, los problemas NP son problemas donde la paralelización tiene poco efecto, dado que al aumentar el tamaño del parámetro recibido, toda la ganancia de la distribución de cargas se desvanece dado que el problema es mucho más grande.
En la figura 5.1 de la página 76, podemos ver ejemplos de diferentes problemas

[1] Se asume logaritmo en base 2

[2] Paralelización es el hecho de poder distribuir un algoritmo para que se ejecute en diferentes procesadores simultáneamente, de modo que en teoría corre más rápido, es equivalente a poner a más personas a hacer el mismo trabajo.

[3] El número de pasos que se pueden ejecutar simultáneamente indicará la cantidad de procesadores que pueden ejecutarse, es falso que si utilizo dos procesadores para una tarea, el tiempo resultante sería la mitad del tiempo que duraría sólo un procesador. Esto debido a la coordinación que tiene que haber entre los procesos. Si al lector le interesa este tema, se profundiza en el capítulo 6. También se recomienda estudiar la Ley de Amdahl

P y NP. Dentro de los problemas P están $O(og(n))$,$O(n)$ y $O(n^2)$ y dentro de los problemas NP están $O(2^n)$ y $O(n!)$.

5.3. Ejemplos de algoritmos con distintas eficiencias

Para poder explicar mejor lo indicado hasta ahora, es importante entender cómo afecta un algoritmo mal diseñado el número de pasos total. A continuación se seleccionaron dos ejemplos concretos sobre cómo diferentes algoritmos que generan el mismo resultado pueden ser completamente diferentes en eficiencia.

5.3.1. El coeficiente binomial

¿Qué es el coeficiente binomial?

Quizá si el lector conoce la historia general y conceptos básicos del coeficiente binomial, entonces probablemente desee saltarse esta parte, pero en caso contrario, es importante explicar lo que es el coeficiente binomial para conocer su importancia y sus aplicaciones.

El coeficiente binomial es el número de cuántos subconjuntos[4] de k elementos puedo formar a partir de un conjunto de n elementos.
Por ejemplo, se tiene el conjunto $\{a, b, c, d\}$ cuya cardinalidad[5] es 4, y queremos obtener la cantidad de formas diferentes en como podemos seleccionar 2 elementos de ese conjunto (recordemos que el orden no importa, es decir, es lo mismo ab que ba). Veamos cuáles son esas formas: ab,ac,ad,bc,bd y cd. Es decir, el coeficiente binomial de $(4, 2)$ es 6.

La forma tradicional de representar el coeficiente binomial es: $\binom{n}{k}$, para el ejemplo anterior, sería $\binom{4}{2} = 6$

El coeficiente binomial recibe su nombre porque indica el coeficiente numérico para el k-esimo sumando de la sucesión obtenida al extender la expresión $(a + b)^n$. De modo que las llamadas "fórmulas notables"se pueden escribir de la siguiente forma:

$$(a + b)^n = \sum_{i=0}^{n} \binom{n}{i} a^{n-i} b^i$$

[4]Es importante recordar que en un conjunto el orden de los elementos es irrelevante (es decir, da lo mismo ab que ba), es por esto que el coeficiente binomial también es denominado *combinaciones*

[5]La cardinalidad es la cantidad de elementos

Formas de calcular el coeficiente binomial

A continuación se muestran diferentes fórmulas para calcular el coeficiente binomial, naturalmente cada una tiene diferente número de pasos.[6]

1. $\binom{n}{k} = \begin{cases} 1 & si \quad k = 0 \\ 1 & si \quad n = k \\ \binom{n-1}{k} + \binom{n-1}{k-1} & en \quad otro \quad caso \end{cases}$

2. $\binom{n}{k} = \begin{cases} 1 & si \quad k = 0 \\ 1 & si \quad n = k \\ \sum_{i=0}^{k} \binom{n-(i+1)}{k-i} & en \quad otro \quad caso \end{cases}$

3. $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ Esta es la más tradicional de todas

4. $\binom{n}{k} = \begin{cases} 1 & si \quad k = 0 \\ 1 & si \quad n = k \\ \frac{n-(k-1)}{k} \times \binom{n}{k-1} & en \quad otro \quad caso \end{cases}$

5. $\binom{n}{k} = f(\binom{n}{k}, 1)$

   $f(\binom{n}{k}, X) = \begin{cases} X & si \quad k = 0 \\ X & si \quad n = k \\ f(\binom{n}{k-1}, \frac{n-(k-1)}{k} \times X) & en \quad otro \quad caso \end{cases}$

   Esta es la versión de recursión de cola de la anterior

Ejecución de los diferentes algoritmos

En esta sección se va a mostrar la ejecución de las diferentes fórmulas del coeficiente binomial descritas en la sección anterior. Para ello se utilizará en todos los casos $n = 4$ y $k = 2$ a modo de ejemplo y la función principal del coeficiente binomial se denotará con las letras *cb*.
Es muy importante que se note que el número de pasos para este ejemplo no constituyen pruebas formales sobre la notación O de los algoritmos, pero estos ejemplos buscan dar una idea *intuitiva* sobre el comportamiento general de las diferentes funciones. Como ejercicio personal, se pueden hacer ejecuciones adicionales con otros valores de n y k.

[6] Para entender el porqué de cada una de las fórmulas el autor recomienda al lector estudiar por su cuenta el *Triángulo de Pascal*

$$\binom{n}{k} = \begin{cases} 1 & si \quad k = 0 \\ 1 & si \quad n = k \\ \binom{n-1}{k} + \binom{n-1}{k-1} & en \quad otro \quad caso \end{cases}$$

La ejecución de esta función es la siguiente:

```
cb(4,2)=cb(3,1)+cb(3,2)
|               cb(3,2)=cb(2,1)+cb(2,2)
|               |       |       cb(2,2)=1
|               cb(3,2)=cb(2,1)+1
|               |       cb(2,1)=cb(1,1)+cb(1,0)
|               |       |       |       cb(1,0)=1
|               |       cb(2,1)=cb(1,1)+1
|               |       |       cb(1,1)=1
|               |       cb(2,1)=1+1
|               cb(3,2)=2+1
cb(4,2)=cb(3,1)+3
|       cb(3,1)=cb(2,1)+cb(2,0)
|       |               cb(2,0)=1
|       cb(3,1)=cb(2,1)+1
|       |       cb(2,1)=cb(1,1)+cb(1,0)
|       |       |       |       cb(1,0)=1
|       |       cb(2,1)=cb(1,1)+1
|       |       |       cb(1,1)=1
|       |       cb(2,1)=1+1
|       cb(3,1)=2+1
cb(4,2)=3+3
6
```

En este caso, para $n = 4$ y $k = 2$ se pudo observar que se necesitaron 20 pasos.

$$\binom{n}{k} = \begin{cases} 1 & si \quad k = 0 \\ 1 & si \quad n = k \\ \sum_{i=0}^{k} \binom{n-(i+1)}{k-i} & en \quad otro \quad caso \end{cases}$$

La ejecución de esta función es la siguiente:

```
cb(4,2)=cb(3,2)+cb(2,1)+cb(1,0)
|                       cb(1,0)=1
cb(4,2)=cb(3,2)+cb(2,1)+1
|               cb(2,1)=cb(1,1)+cb(0,0)
|               |               cb(0,0)=1
|               cb(2,1)=cb(1,1)+1
|               |       cb(1,1)=1
|               cb(2,1)=1+1
cb(4,2)=cb(3,2)+2+1
|       cb(3,2)=cb(2,1)+cb(1,0)
|       |               cb(1,0)=1
|       cb(3,2)=cb(2,1)+1
|       |       cb(2,1)=cb(1,1)+cb(0,0)
|       |       |               cb(0,0)=1
|       |       cb(2,1)=cb(1,1)+1
|       |       |       cb(1,1)=1
|       |       cb(2,1)=1+1
|       cb(3,2)=2+1
cb(4,2)=3+2+1 **dos pasos
6
```

En este caso, para $n = 4$ y $k = 2$ pudimos observar que se necesitaron 19 pasos. Consideremos que la suma de más de dos sumandos, necesita un paso adicional por cada sumando extra que se está calculando.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Esta es la versión más corta de todas y es, de las vistas en este capítulo, la única que no está definida de forma recursiva.La ejecución es muy sencilla, tal y como se muestra a continuación:

```
             4!         1x2x3x4       24
cb(4,2)=----------- = --------- = ------ = 6
          2!(4-2)!      1x2(1x2)      4
```

Es muy importante notar que la cantidad de pasos en este caso se cuenta un poco distinto. Se tienen 4 pasos para calcular el numerador, 4 para el denominador y un paso adicional para la división, en total fueron 9 pasos.

$$\binom{n}{k} = \begin{cases} 1 & si \quad k = 0 \\ 1 & si \quad n = k \\ \frac{n-(k-1)}{k} \times \binom{n}{k-1} & en \quad otro \quad caso \end{cases}$$

Esta versión es claramente recursiva, a continuación, su proceso de ejecución:

```
         3
cp(4,2)=---x cp(4,1)
|        2           4
|            cp(4,1)=---x cp(4,0)
|            |        1
|            |            cp(4,0) = 1
|            |        4
|            cp(4,1)=---x 1
|                     1
|        3     4
cp(4,2)=--- X ---
         2     1
6
```

Dependiendo de como se manejen las fracciones (ya sea dividiendo en cada oportunidad o dividiendo hasta el final (mucho mejor al disminuir los pasos), la cantidad de pasos total del algoritmo pueden variar. Suponiendo que se divide hasta el final, este algoritmo pudo resolver el $\binom{4}{2}$ en 6 pasos.

$$\binom{n}{k} = f(\binom{n}{k}, 1)$$

$$f(\binom{n}{k}, X) = \begin{cases} X & si \quad k = 0 \\ X & si \quad n = k \\ f(\binom{n}{k-1}, \frac{n-(k-1)}{k} \times X) & en \quad otro \quad caso \end{cases}$$

Tal y como se indicaba en el capítulo 4 el uso de la recursión de cola tiende a disminuir la cantidad de pasos de las funciones recursivas. Aunado a lo visto en este capítulo, está de más indicar que la recursión de cola es mucho más eficiente dado que utiliza menos pasos. Veamos la ejecución de esta función:

```
cb(4,2)=cb'(4,2,1)
        cb'(4,1,3/2)
        cb'(4,0,6)
        6
```

Tomando en cuenta la división final, el problema tomó 5 pasos en resolverse.

Como hemos visto, cada uno de estos algoritmos llegan al mismo resultado. Se puede demostrar matemáticamente que todos los algoritmos son equivalentes en ese aspecto. Sin embargo, de manera empírica se ha demostrado que el número de pasos varía según el algoritmo seleccionado, consecuentemente el tiempo de respuesta también va a variar.

5.3.2. Potencia: Elevar una base a un exponente entero

El hecho de hacer los cálculos necesarios para b^e tiene muchas aristas. Veamos una primer versión iterativa:

```
potencia(base,exp)
    contador<-0
    resp<-1
    mientras(contador<exp)
        resp<-resp*base
        contador<-contador+1
    retornar resp
```

Suponga que se tiene como base 3 y como exponente 58. Este algoritmo utilizaría 58 pasos para poder calcular el valor de 3^{58}, por razones de espacio se omitirá la ejecución del algoritmo completo, pero se espera que el lector pueda apreciar que ese algoritmo tiene $O(n) = n$ (donde n es el exponente)

Consideremos la siguiente propiedad de las potencias:

$$a^n = \begin{cases} (a^{\frac{n}{2}})^2 & si \quad n \quad par \\ a \times (a^{\frac{n-1}{2}})^2 & en \quad otro \quad caso \end{cases}$$

Se trata de algo muy sencillo, por propiedades de potencias, 5^8, por ejemplo, es lo mismo que decir $5^4 \times 5^4$. Entonces aprovechando esta característica,

podemos definir la siguiente fórmula recursiva:

$$c(n) = n \times n$$

$$f(base, exp) = \begin{cases} 1 & si \quad exp \quad = 0 \\ cuad(f(base, \frac{exp}{2}) & si \quad exp \quad par \\ base \times cuad(f(base, \frac{exp-1}{2}) & en \quad otro \quad caso \end{cases}$$

A continuación se detalla la ejecución para una base 3 y un exponente 58:

```
f(3,58)=c(f(3,29))
          f(3,29)=3(c(f(3,14)))
                     f(3,14)=c(f(3,7))
                                f(3,7)=3(c(f(3,3)))
                                           f(3,3)=3(c(f(3,1)))
                                                      f(3,1)=3(c(f(3,0)))
                                                                 f(3,0)=1
                                                      f(3,1)=3(1)
                                           f(3,3)=3(c(3))
                                           f(3,3)=3(9)
                                f(3,7)=3(c(27))
                                f(3,7)=3(729)
                     f(3,14)=c(2187)
                     f(3,14)=4782969
          f(3,29)=3(c(4782969))
          f(3,29)=3(2.2876792e+13)
f(3,58)=c(6.8630377e+13)
f(3,58)=4.7101287e+27
```

Más allá del abultado resultado, es importante notar la cantidad de pasos que se necesitaron para obtenerlo. Este segundo algoritmo, es del orden $O(n) = log(n)$ donde n es el valor del exponente. Si bien es cierto, el número de pasos es mayor a $log_2(58)$, en este caso, se tiene que contar el camino de ida ($O(n) = log(n)$), el de vuelta ($O(n) = log(n)$) y elevar al cuadrado ($O(n) = 1$), que suman pasos, sin embargo, el orden de crecimiento sigue siendo $O(n) = log(n)$.

5.3.3. El vendedor viajero: un problema NP

En esta sección se han desarrollado diferentes algoritmos con diferentes $O(n)$, siempre buscando que los ejemplos sean significativos en la comprensión de la relación que existe en el número de pasos para poder resolver el problema.

El vendedor viajero es uno de los problemas cuyo $O(n)$ es más complicado de resolver, en su forma pura se trata de un problema $n!$, aunque hay aproximaciones cuyo $O(n) = 2^n n^2$.

Descripción del problema

Imaginemos que un vendedor quiere vender sus productos en n ciudades. Asumamos que cada una de estas ciudades tiene un tren (es completamente similar si se imagina que son aviones o buses) que va a cada una de las $n-1$ ciudades restantes. Cada tiquete de tren tiene un costo diferente, dependiendo de la demanda y de la distancia entre las ciudades (como es normal).
El amigo vendedor, desea poder visitar todas las ciudades y vender su producto en cada una de ellas. Para ello debe de viajar entre las ciudades, sin embargo, para poder maximizar sus ganancias, debe minimizar sus gastos, por lo tanto desea poder saber cuál es la ruta que tiene que tomar, para que el costo de traslados entre ciudades sea el menor.

Principio de conteo de rutas

El problema principal, consiste en que se tienen que calcular todas las rutas. El costo de viajar en una ruta " x ", lo que toma son n pasos, dado que son n ciudades y recorrer la ruta implica sumar los costos de traslado entre una ciudad y otra.
Sin embargo, la cantidad de rutas sí es muy significativa. Suponga que tiene n ciudades. Para visitar la primera ciudad, tiene n posibilidades, para visitar la segunda ciudad tiene $n-1$ posibilidades, para visitar la tercera ciudad tiene $n-2$ posibilidades, etc.

Veamos:

$\underline{n} \times \underline{n\text{-}1} \times \underline{n\text{-}2} \times \ ... \ \times \underline{2} \times \underline{1} = n!$

Esto demuestra que se tratan de $n!$ rutas.

En la siguiente tabla se muestra un pequeño ejemplo para diferentes valores de n y las todas las posibles rutas para esa cantidad.

n	**Conjunto de ciudades**	**Rutas**
1	{A}	no hay rutas
2	{A,B}	AB BA
3	{A,B,C}	ABC ACB BAC BCA CAB CBA
4	{A,B,C,D}	ABCD ABDC ACBD ACDB ADBC ADCB BACD BADC BCAD BCDA BDAC BDCA CABD CADB CBAD CBDA CDAB CDBA DABC DACB DBAC DBCA DCAB DCBA

Como se puede apreciar la cantidad de rutas sigue una distribución factorial (salvo cuando se trata de una sola ciudad, para cuyo caso, el problema carece de sentido). El calcular cada una de estas rutas, aunque pudieran ser calculadas en un sólo paso, habría que hacer esa operación $n!$ veces, por lo que el problema se convierte en uno de los problemas de más difícil solución.

5.4. Como calcular la complejidad temporal

Hasta el momento se ha mostrado, de forma muy general, la importancia de la cantidad de pasos de un algoritmo y cómo es que diferentes algoritmos que tienen el mismo resultado, pueden tener diferente número de pasos.

Sin embargo, dado un algoritmo, es importante determinar a partir del algoritmo el número de pasos que está siendo descrito.

5.4.1. Suma de los primeros n números naturales

Supongamos que queremos calcular 1+2+3+4+5+6+...+n, el algoritmo está descrito en el capítulo 3 (de forma ineficiente)

```
suma(n)
    i<-0                % 1 paso
    total<-0            % 1 paso
    mientras(i<n)       %
        total<-total+i  % n pasos
        i<-i+1          % n pasos
    retornar total      % 1 paso
                  %%TOTAL: 2n+3 pasos
```

Como vemos la cantidad de pasos que este algoritmo requiere son $2n+3$, sin embargo, es $O(n)$, esto se da porque la notación O indica la forma en como **crece** la función, obviando las constantes numéricas, es decir, el nivel de crecimiento conforme el tamaño de la entrada n aumenta, es decir, no indica el número exacto de pasos que tiene la función o algoritmo, sino la forma como se graficaría, tendiendo el tamaño a infinito. La manera para saber empíricamente[7] el O consiste en tomar el polinomio con el número de pasos "exacto"[8] del algoritmo, tomamos el monomio de mayor grado y eliminamos los coeficientes numéricos, así por ejemplo, en: $2n+3$, se tienen dos monomios, el primero de grado 1 y el segundo de grado 0, como vemos el monomio $2n$ es el de mayor grado, y una vez que eliminamos los coeficientes numéricos (el 2), queda que $O(n)$.

La suma de cuadrados utilizando la referencia de Gauss

El algoritmo anterior se puede simplificar bastante si pudiéramos considerar lo siguiente:

$$S = 1 + 2 + 3 + ... + (n-2) + (n-1) + n$$

Por tanto, dado que la suma es conmutativa,

$$S = n + (n-1) + (n-2) + ... + 3 + 2 + 1$$

Si sumamos las dos ecuaciones queda:

$$2S = (n+1) + (n+1) + (n+1) + ... + (n+1) + (n+1) + (n+1)$$

[7] Es importante que el lector sepa que existen demostraciones matemáticas formales para determinar O de un algoritmo, sin embargo, se omiten de este libro por su carácter introductorio

[8] Los pasos de un algoritmo no necesariamente corresponden con los pasos ejecutados en una computadora, factores como el diseño de la computadora y el lenguaje de programación afectan directamente la cantidad de instrucciones que se ejecutan, sin embargo, la mejor referencia que se tiene para saber la cantidad de pasos necesaria, es el algoritmo, el cual no cambia independientemente de la arquitectura y el lenguaje

Por tanto, vemos que $2S$ es en realidad, la suma de sólo $n+1$, en este caso particular, podemos apreciar que son n sumandos, por lo que:

$$2S = n \times (n+1)$$

lo que significa que

$$S = \frac{n(n+1)}{2}$$

Convirtiendo eso a fórmula:

```
suma(n)
   retornar (n * (n+1))/2      % 1 paso
```

Dado que en este caso, el número de pasos es independiente del tamaño de la entrada, entonces se considera que es $O(1)$, porque el número de pasos es **constante**.

5.5. Problemas

Aunque la eficiencia y el $O(n)$ será un tema recurrente en los capítulos venideros y por ende en los problemas de los demás capítulos, los ejercicios de este capítulo busca darle al lector una mejor idea de las implicaciones que tiene el número de pasos y su importancia.

5.5.1. Programando y probando el coeficiente binomial

Programe cada uno de las formas vistas en este capítulo del coeficiente binomial.
Se invita al lector a hacer pruebas de rendimiento con diferentes valores de n y k para que pueda ver la eficiencia de cada uno de los algoritmos.

5.5.2. Calculando O

Suponga que los siguientes polinomios corresponden con los pasos de diversos algoritmos, indique la función en notación O de cada uno de ellos. En caso de ser necesario, simplifique el polinomio.

1. $5n + 7$
2. $7n^2 + 3n - 7$
3. $2n^3 + 7^{23}n + 2$
4. $\frac{n}{2000!} + 9$
5. $\frac{7n^4}{2n} + 2^9n^2 + 8n + 3$
6. $\frac{15n^5}{101n^2} + \frac{8n^4}{55n} + 600n^3$

7. $2\sqrt{n} + 8$
8. $log_2 n^n + 100!$
9. $2^n n^3 + n!$
10. $\frac{2^n}{n} + \sum_{i=0}^{100}(i+1)n^i$

5.5.3. Análisis

Para cada uno de los siguientes algoritmos, dejando por fuera su función, indique el O de ellos. Considerar que $<-$ es una asignación, equivalente a $=$ en muchos lenguajes de programación.

1.
```
f1(n)
      nvo<-n/2
      resp <- si
      mientras(nvo > 1)
          si (n mod nvo = 0)
              resp <- no
              nvo <- 1
          nvo <- nvo-1
      retornar resp
```

2.
```
f2(n)
       i<-1
       mientras(i x i < n)
        i <- i+1
       retornar ixi = n
```

3.
```
f3(n)
       i<-n
       mientras(i>1)
            j <- 1
            mientras(j<n)
                 j <- j+1
            i <- i/2
```

4.
```
f4(n)
       i <- n
       j <- n
       resp <- 0
       mientras (i>0)
            mientras (j>0)
                 resp<-resp+i+j
                 j <- j-1
            j <- n
```

```
            i <- i-1
        retornar resp
```

5.
```
f5(n)
      si (n = 0)
          retorna 0
      si (n = 1)
          retorna 5
      si (n = 2)
          retorna 7
      f4(n-1)+f4(n-2)+f4(n-3)
```

5.5.4. Mirando atrás

Retome los ejercicios programados del capítulo 3, en la subsección 3.7.3 de Ciclos. Para cada uno de los problemas resueltos, indique el $O(n)$ de cada uno de ellos.

5.6. Resumen

El presente capítulo trabajó el tema de eficiencia, definido como el número de pasos necesario para completar un algoritmo con base en el tamaño de la entrada. De manera muy informal, se definieron algunos conceptos asociados y se mostró una tabla con los tiempos supuestos bajo cierto marco de hardware como referencia.

Los ejemplos mostrados, donde de varias formas se llega a la misma conclusión, son una muestra clara de como es que la eficiencia se convierte en la "vanidad" del programador y esta debe ser una prioridad de los buenos programadores en cada problema que desarrolle.

En capítulo 6 se mostrará como se utilizan estructuras para facilitar el manejo de conjuntos de datos.

Capítulo 6

Listas, vectores y matrices

La sala se torna oscura
se escucha un ruido muy suave,
y en este ambiente tan grave
brota un haz de luz muy pura.
De inmediato, ya segura,
sobre una pantalla blanca
se muestra una imagen franca.
Fotos son en movimiento,
que te mantienen atento
desde que la historia arranca.(...)

Arnaldo Perez Portela

Hasta el momento se han visto problemas de cálculo numérico, sin embargo, una computadora también es capaz de almacenar grupos de números y es posible hacer algoritmos que trabajen con dichos grupos de números.

Tal y como se vio en el capítulo 2, muchos datos que se muestran en una computadora, tales como sonidos e imágenes, están relacionados con estos conceptos, que son plasmados por el autor cubano referenciado al inicio de este capítulo.

Este capítulo está destinado a desarrollar algunos de estos algoritmos.

6.1. Diferencias entre lo estático y lo dinámico

Hasta el momento se ha señalado que una computadora puede trabajar sobre grupos de datos, sin embargo, hay dos formas principales de almacenar estos grupos de datos en memoria[1], una es separando un espacio de memoria de tamaño fijo (o estática) y otra es tomando un pequeño espacio de memoria y separándolo en dos, por un lado el dato que se desea almacenar y por otro una

[1] En el capítulo 2 se explica brevemente lo que es la memoria en una computadora

dirección de otra pequeña parte de memoria que posee también una distribución similar, teniendo una estructura de un tamaño desconocido a priori (o dinámica).

6.1.1. Memoria Estática

Lo estático normalmente tiene ciertas ventajas sobre lo dinámico en cuanto a facilidad para escribir programas, aunque tiende a tener ciertas restricciones de tamaño, estas desventajas se ven compensadas porque simplifican significativamente el desarrollo de algoritmos.

También permite acceder a los campos de manera directa, esto es posible porque se conoce la dirección desde donde comienza la sección de memoria y se conoce el número de bytes que hay antes de la sección deseada, por lo que la dirección del lugar al que quiera llegar es la dirección inicial más la cantidad de bytes que debe moverse. Por su parte, el uso de memoria dinámica obliga a recorrer toda la estructura para poder dar con la meta.

6.1.2. Memoria Dinámica

Lo dinámico también posee ventajas sobre lo estático (si no fuera así, probablemente no se estudiaría). Las ventajas de los dinámico son las desventajas de lo estático y viceversa. En primer lugar es que el tamaño para almacenar datos es potencialmente infinito (claro que va a depender del tamaño físico de la memoria de la computadora). Por otra parte permite montar estructuras de datos mucho más complejas y en muchas ocasiones necesarias para lograr un acceso más rápido. Tenemos por ejemplo que para poder tener acceso a ciertas partes del disco se utiliza memoria dinámica. Un sistema de archivos, entre otras cosas, permite crear, borrar y modificar archivos dinámicamente.

6.2. Vectores

Un vector es representado en una computadora como segmento estático de memoria, pero es mucho más que eso. Por ejemplo en dos dimensiones se tienen puntos típicamente denominados (x,y) pero ese par ordenado, en sí mismo es un vector de números de dos campos.

Un hecho importante es que el vector posee dirección en el espacio dimensional en que se encuentra, así un vector de dos elementos tiene una dirección en un plano, si el vector tiene tres elementos, entonces tiene una dirección en un espacio tridimensional y si tiene más, aún así se puede estudiar su espacio vectorial, como una dirección en ese espacio n-dimensional.

6.2.1. Operaciones con vectores

En el mundo de la matemática se pueden realizar diversas operaciones con vectores. Para poder trabajar sobre ellos, se trabaja sobre las posiciones que ocupan los elementos en el vector.

Así el vector a = {10,12,33}, podemos verlo como el conjunto de posiciones $a_1 = 10$, $a_2 = 12$ y $a_3 = 33$. Por razones físicas en computación, un vector de n elementos en lugar de numerarlos de la posición 1 a la n, se numera de la posición 0 a la $n - 1$, pero la idea es poder aprovechar todos los campos del espacio físico la computadora otorga.

De esta manera y en honor a que este es un libro de fundamentos de programación, se utilizará de ahora en adelante la forma de numeración tradicional en la computación, por lo que el vector a = {10,12,33} será entonces $a_0 = 10$, $a_1 = 12$ y $a_2 = 33$.

Otro detalle, siempre tiene que haber una manera de calcular la cantidad de elementos de un vector, es decir, el largo del vector.

Escalar un vector

Uno de las primeras operaciones que existen sobre vectores es la de escalarlos, lo que consiste en multiplicar todos sus elementos por un mismo número.

Es decir $\forall i, a_i = e * a_i$[2]

Un algoritmo para representar lo anterior, sería:

```
escalarVector (valor e, vector v)
     i <- 0
     largo <- largo(v)
     mientras(i<largo)
      v[i] <- v[i]*e
      i <- i+1
```

Ejemplo

Se tiene el vector {34,132,44} y el valor escalar 0.5, una vez que se multiplica el vector por el escalar, el vector queda "escalado" en {17,66,22}.

Sumar dos vectores

La suma de vectores es algo mucho más sencillo, simplemente, se tienen dos vectores cuyo largo debe ser igual[3] y posición por posición, obtener la suma de cada uno de sus elementos. Quizá lo interesante de este problema consiste en que se necesitan tres vectores, los dos sumandos y el vector donde se colocará la respuesta. Desde el punto de vista matemático la fórmula sería así: $\forall i, s_i = v1_i + v2_i$ Entonces el algoritmo sería (sin validar que sean iguales):

```
sumaVectores (vector v1, vector v2)
     i <- 0
     largo <- largo(v1)
     s <- NuevoVector de tamaño largo
     mientras(i<largo)
      s[i] <- v1[i]+v2[i]
      i <- i+1
```

[2]El signo ∀ significa "para todo"

[3]Resulta imposible sumar dos vectores dimensionalmente diferentes

Ejemplo
Supongamos que tenemos los vectores {1,2,3} y {4,-3,-10}. La suma de ellos es: {5,-1,-7}

Producto Punto

Una de las primeras curiosidades que se encuentran a la hora de trabajar con vectores, es que el multiplicar dos vectores, da como resultado un escalar, es decir, un número.

La fórmula para calcular el producto de vectores, donde una vez más, sólo se puede calcular si tienen el mismo largo, es: $\sum_{i=0}^{n} v1_i * v2_i$

Ejemplo
Supongamos que se tiene el vector {2,4,8} y el vector {3,6,9}, el producto de estos dos vectores es: $2 \times 3 + 4 \times 6 + 8 \times 9 = 102$

Norma del vector

La norma del vector está dada por la raíz cuadrada de la suma de los diversos elementos al cuadrado. Es decir:

$$\sqrt{\sum_{i=0}^{n-1} v_i^2}$$

Tradicionalmente la norma del vector v se define como $|v|$

Ejemplo
Supongamos que se tiene el vector {3,4}, entonces su norma es 5, porque: $\sqrt{3^2 + 4^2} = 5$. O por ejemplo, también podemos considerar el vector {5,5,5,5} la norma es 10, esto porque $\sqrt{5^2 + 5^2 + 5^2 + 5^2} = 10$

Ángulo entre vectores

Los vectores representan una dirección desde un eje (típicamente en 0), entonces se puede calcular el ángulo entre ellos, partiendo de ese eje. Para poder encontrarlo, basta con despegar α de la siguiente ecuación: $vu = |u| \times |v| \times cos(\alpha)$, donde u es un vector y v es el segundo vector.

Ejemplo
La manera más fácil de ver este efecto es sobre vectores muy sencillos de dos dimensiones (por lo tanto que sólo tienen dos elementos) en un plano de coordenadas cartesianas.

6.3. Matrices

Las matrices son en el fondo, vectores de vectores. Es decir, se representan como una cuadrícula. A continuación se muestran diversas matrices:

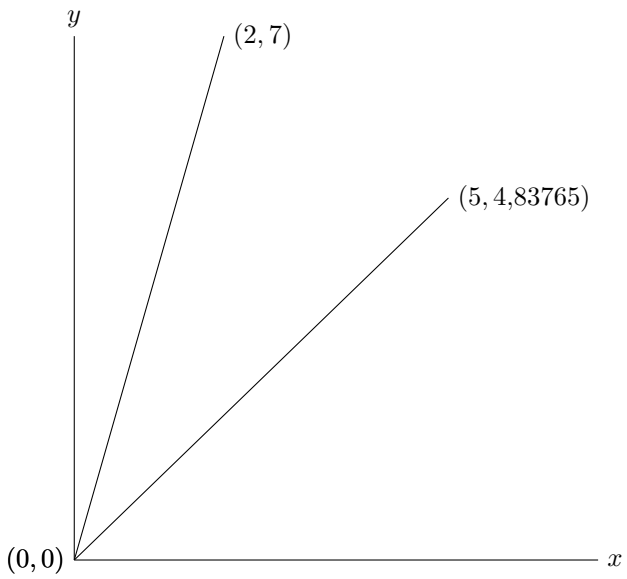


Figura 6.1: Dos vectores en ángulo de 30°

$$a = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$
$$b = \begin{pmatrix} b_{11} & b_{12} \end{pmatrix}$$
$$c = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$
$$d = \begin{pmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & d_{34} \end{pmatrix}$$
$$e = \begin{pmatrix} e_{11} \\ e_{21} \\ e_{31} \end{pmatrix}$$

Como se puede apreciar, un vector, también es una matriz, pero puede ser vector fila, como el caso de b o vector columna, como en el caso de e. Además, los elementos en la matriz son ubicados por posiciones fila y columna.

Definiciones

1. Fila: Son los vectores horizontales de la matriz.
2. Columna: Son los vectores verticales de la matriz.
3. Diagonal: Una diagonal es el vector de elementos que se encuentran desde la esquina superior izquierda hasta la esquina inferior derecha.

4. Matriz Triangular Superior: Es una matriz cuadrada que tiene bajo la diagonal, únicamente el número 0 (cero).

5. Matrizu Triangular Inferior: Es una matriz cuadrada que tiene sobre la diagonal úicamente el número 0 (cero).

6. La matriz identidad es una matriz de $n \times n$ que tiene en su diagonal compuesta por el número 1 (uno) y es triangular superior e inferior a la vez. Tiene la propiedad que si es multiplicada por cualquier matriz M, el resultado es M.

6.3.1. Operaciones con matrices

Al igual que los vectores, las matrices se pueden operar. A continuación se muestran algunas de las operaciones fundamentales de una matriz.

Suma de matrices

La primera operación que se puede realizar es la suma. La condición es que ambas matrices tengan el mismo largo y el mismo ancho. Incluso la matriz resultante, tendrá las mismas dimensiones.

La forma de la suma es bastante sencilla, supongamos que se tienen las matrices A y B, van a ser sumadas en la matriz C. Se tiene que cumplir que $c_{ij} = a_{ij} + b_{ij}\ \forall_{i,j}$

Ejemplo

$$\begin{pmatrix} 2 & 23 & 3 & -7 \\ 3 & \pi & -3 & 14 \\ 2 & 31 & 13 & 5 \end{pmatrix} + \begin{pmatrix} -42 & 23 & 32 & 7 \\ 34 & \pi & -6 & 55 \\ 9 & 28 & -130 & 5 \end{pmatrix} = \begin{pmatrix} -40 & 46 & 35 & 0 \\ 37 & 2\pi & -9 & 69 \\ 11 & 59 & -117 & 10 \end{pmatrix}$$

Transposición de matrices

La transposición de matrices consiste en tomar una matriz y cambiar los vectores fila y pasarlos a ser vectores columna. Es decir, se trata de tomar una matriz y "girarla" 90 grados. Quizá un dato curioso, que si A es una matriz cuadrada, entonces su diagonal es igual a la de A transpuesta (A^T).

Ejemplo

$$A = \begin{pmatrix} 2 & 23 & 3 & -7 \\ 3 & \pi & -3 & 14 \\ 2 & 31 & 13 & 5 \end{pmatrix} -> A^T = \begin{pmatrix} 2 & 3 & 2 \\ 23 & \pi & 31 \\ 3 & -3 & 13 \\ 7 & 14 & 5 \end{pmatrix}$$

Multiplicación de matrices

El porqué de la multiplicación sea de esta forma quizá sea un poco complicado de explicar para un texto introductorio como este, sin embargo, es impor-

tante que el lector o lectora comprenda que se trata de un cruce de vectores n-dimensionales por vectores m-dimensionales.

A continuación se mostrará el proceso de la multiplicación, suponga que se tiene una matriz A de $m \times n$ y una matriz B de $n \times k$, por lo tanto $A \times B = C$ implicaría que C es de $m \times k$.

De manera que se cumpla que: $c_{ij} = a_i \times c\_{}_j \ \forall i < m, j < k$
De manera que la posición ij de la matriz resultante es el producto punto del vector fila i de A, por el vector columna j.

Ejemplo

$$\begin{pmatrix} 2 & 2 & 3 & -7 \\ 3 & 4 & -3 & 4 \end{pmatrix} \times \begin{pmatrix} 2 & 2 & 4 \\ 3 & 1 & 1 \\ 2 & 0 & 5 \\ 4 & 3 & 0 \end{pmatrix} = \begin{pmatrix} -12 & -15 & 25 \\ 28 & 22 & 1 \end{pmatrix}$$

Operaciones elementales

Muchos algoritmos se simplifican al modelarlos mediante una matriz. Muchos de estos algoritmos utilizan ciertas operaciones particulares sobre las filas o columnas de la matriz, estas operaciones son denominadas operaciones elementales.

Las operaciones elementales lo que buscan es principalmente cambiar los vectores fila o columna de una matriz al sumarle el producto de un escalar por otro vector fila.

Ejemplo

$$\begin{pmatrix} 2 & 3 & 2 \\ 4 & 9 & 6 \\ 3 & -3 & 13 \\ 7 & 14 & 5 \end{pmatrix}$$

A esa matriz puedo hacerle una operación elemental que consiste en dividir la fila 1 por -2 (que es lo mismo que multiplicar por $\frac{-1}{2}$) y sumar ese resultado en la misma fila 1). La notación de esta operación sería: $\overline{F1/2}$, donde la línea encima de la expresión indica en qué fila debe escribirse el vector resultante. Dejando la matriz anterior de la siguiente manera:

$$\begin{pmatrix} 1 & \frac{3}{2} & 1 \\ 4 & 9 & 6 \\ 3 & -3 & 13 \\ 7 & 14 & 5 \end{pmatrix}$$

Otra operación fundamental sería $-4F1 + \overline{F2}$ que sería multiplicar la fila 1 actual por -4 y sumar ese resultado a la fila 2. Si aplicamos a la matriz anterior esa operación, el quedaría modificada de la siguiente manera:

$$\begin{pmatrix} 1 & \frac{3}{2} & 1 \\ 0 & 3 & 2 \\ 3 & -3 & 13 \\ 7 & 14 & 5 \end{pmatrix}$$

6.3.2. Exponenciación logarítmica de matrices

Tal como se vio en el capítulo 5, propiamente en la sección 5.3.2, existe una manera de calcular la exponeciación o potencia de un número de manera mucho más rápida que multiplicando por sí mismo repetidas veces. Ese mismo proceso se puede aplicar para elevar o exponenciar una matriz a un entero. Siendo M una matriz *cuadrada*, se cumple lo siguiente:

$$M^n = \begin{cases} (M^{\frac{n}{2}})^2 & si \quad n \quad par \\ M \times (M^{\frac{n-1}{2}})^2 & en \quad otro \quad caso \end{cases}$$

Usando la matriz identidad como caso base para $n = 0$.

Ejemplo de uso de exponeciación logarítmica de matrices

Un uso muy común de lo explicado en la sección anterior, es el poder calcular un elemento de una función recursiva simple, pero mucho más rápido. Tomemos el caso de Fibonacci:

$$fibo(n) = \begin{cases} 1 & si \quad n = 1 \\ 1 & si \quad n = 2 \\ fibo(n-1) + fibo(n-2) & en \quad otro \quad caso \end{cases}$$

Resulta que: $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \times \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} fibo(n) \\ fibo(n+1) \end{pmatrix}$

La matriz característica (la que es de 2×2) se forma con coeficientes particulares, las primeras $n-1$ filas, tienen como primer columna sólo ceros y a partir de la segunda columna, deben formar una matriz identidad de $n-1 \times n-1$. La última fila, tiene los coeficientes de la función recursiva.

Se le invita al lector a programar fibonacci de esta otra manera y probar efectivamente la eficiencia a la que se ve expuesto.

6.4. Listas

Las listas son escencialmente dinámicas, es decir no se representan por un espacio acotado de memoria, sino que en principio pueden crecer de manera indeterminada, esto dependiendo de la cantidad de memoria que se tenga disponible en la computadora, pero algorítmicamente hablando, podrían ser potencialmente infinitas.

Dependiendo del lenguaje de programación seleccionado el manejo de las listas diferirá dependiendo de si éstas son parte esencial del lenguaje o no, es decir si ya existe un tipo lista en el lenguaje o no. Sin embargo, las operaciones aquí descritas se limitan a descripciones algorítmicas. Para mayor detalle en la descripción propia de las listas dependiendo de si se trata de lenguajes no declarativos, se puede remitir al apéndice B donde se describirá en los diferentes lenguajes cómo implementar la lista.

Sin embargo, sin importar como se implementan o el lenguaje de programación utilizado, siempre la representación en una computadora es la misma. En su representación más simple, las listas tienen elementos, los cuales ocupan un espacio de memoria y están "enlazados", de modo que junto con el espacio de memoria donde está el dato o elemento, también se encuentra la dirección del espacio de memoria donde está el siguiente elemento. Al segmento de memoria que comprende el dato y la dirección se le conoce como "nodo", entonces se dice que un nodo es la tupla que contiene dato y un "puntero" al siguiente nodo[4].

Muchos lenguajes de programación (típicamente los declarativos) encapsulan este hecho y permite que se hagan operaciones totalmente ajenas a la implementación de la lista y otros (los estructurados), por el contrario, demandan que sea el programador quien se encargue de todos los pormenores. El lenguaje seleccionado para trabajar debería ser determinado por el nivel de comprensión esperado en relación con la arquitectura de la computadora vs la abstracción para desarrollar algoritmos. Sigue siendo una sugerencia del autor concentrarse primero en los algoritmos y luego, una vez dominado este punto, comprender la implementación en una arquitectura computacional.

A continuación se muestran algunas de las operaciones de listas más comunes, es importante tener presente que se trata de estructuras dinámicas y por lo tanto estas operaciones tienen sentido, pues modifican la estructura general de la lista. Para todos los ejemplos se utilizará la lista enlazada representada en la figura 6.2.

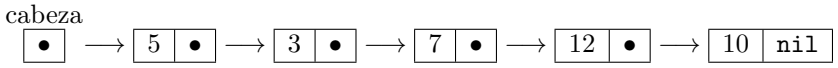


Figura 6.2: Lista Enlazada

6.4.1. Insertar

La inserción es la primera de las operaciones de listas, pues principalmente es la que construye y separa nueva memoria para hacer que la lista crezca.

El insertar tiene diferentes matices, insertar al inicio, al final o en medio, con diferentes condiciones que deben ser validadas, pero una vez más, dependen de la implementación.

Ejemplo

Para este ejemplo se insertará π en la lista. Las listas resultantes se muestran a continuación:

[4]Nótese que el último nodo no tiene ninguna dirección a la cual apuntar, por lo que su espacio de dirección apunta a lo denominado *nulo*, representado por *nil* ó *null*

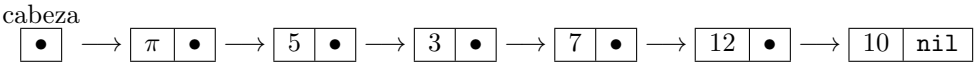


Figura 6.3: Insertar al inicio

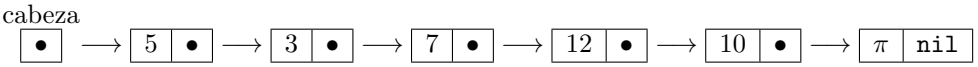


Figura 6.4: Insertar al final

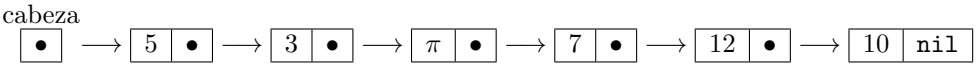


Figura 6.5: Insertar en medio

6.4.2. Eliminar

El eliminar es el proceso inverso a la inserción, en lugar de construir la lista, es por lo tanto, en realidad un proceso de destrucción, donde se busca liberar memoria ocupada por un nodo previamente.

Al igual que en el caso anterior, a una lista se le puede eliminar por el inicio, por el fin o en medio.

Ejemplo

Supongamos que se cuenta con la siguiente lista:

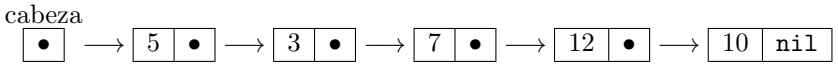


Figura 6.6: Lista Enlazada

6.5. Otras operaciones de listas

Hasta ahora, se han utilizado gráficos con el fin de facilitar la comprensión de los ejemplos, sin embargo, en adelante se seguirá utilizando pseudo-código. Es muy recomendable, en todo caso, hacer los dibujos a mano para comprender el comportamiento del algoritmo.

6.5.1. Largo

El largo de una lista me indica la cantidad de elementos que esta posee. Para ello se debe contar desde el primero hasta el último nodo (el que tiene como siguiente un puntero nulo).

Ejemplo

```
buscar (elem, lista)
     i <- 0
     nodo <- lista.primero   **se asume que así se obtiene
                                la cabeza de la lista
     mientras(nodo <> nil)
      i<-i+1
      nodo<-nodo.sigte   **se asume que así se 'viaja' entre nodos
    retorna i
```

6.5.2. Buscar

Buscar un elemento tiene dos connotaciones, una es saber si una lista contiene un elemento, la otra consiste en saber en qué posición (partiendo de la posición 0) se ha encontrado ese elemento, devolviendo -1 en caso de no haberlo encontrado.

Ejemplo

El algoritmo para el segundo caso (retornar la posición en que se encontró), sería algo así:

```
buscar (elem, lista)
     i <- 0
     nodo <- lista.primero
     mientras(nodo <> nil)
      si(nodo = elem)
      retorna i
      i<-i+1
      nodo <- nodo.sigte
    retorna -1
```

6.6. Problemas

6.6.1. Vectores

1. Máximo

   Recibe un vector y devuelve otro vector de dos posiciones, la primera posición tendrá el valor máximo del vector y la segunda tendrá la posición del máximo en el vector. En caso de haber empate, se toma la primera aparición del valor.

2. Mínimo

   Recibe un vector y devuelve otro vector de dos posiciones, la primera posición tendrá el valor mínimo del vector y la segunda tendrá la posición de ese valor en el vector. En caso de haber empate, se toma primera aparición del valor.

3. Multiplicar escalar por vector

   Dado un número y un vector, retornar todo el vector con cada elemento multiplicado por el escalar.

4. Suma de vectores

   Dados dos vectores, retornar la suma de los vectores. Considere que si $v + u = w \implies \forall i \ w_i = v_i + u_i$. Considerando a los dos vectores del mismo tamaño.

5. Producto punto

   Escriba el producto punto pero utilizando recursión de cola. De igual manera la función debe recibir los dos vectores que va a multiplicar.

6. Solución de un polinomio

   Recibe un vector de enteros, cada uno representa el coeficiente de un polinomio. Por ejemplo, si tengo un vector de la forma : {3,2,6,2}, eso estaría representando un vector de la forma: $3x^3 + 2x^2 + 6x + 2$. Dado ese vector, se debe retornar otro vector con las posibles soluciones que hacen que el polinomio tenga un valor de cero.

7. Distancia entre vectores

   Dados dos vectores, indicar la distancia entre los dos vectores v y w consiste en la siguiente fórmula:
   $S(v, w) = \sqrt{\sum_{i=1}^{n} (v_i - w_i)^2}$
   (siempre y cuando los vectores sean del mismo tamaño).

8. Vector a g grados de otro vector

   Cree un programa que para un vector v en dos dimensiones y un grado g, genere un vector w, naturalmente de dos dimensiones también, de modo que el grado entre v y w sea g. Sugerencia, asuma un valor de x arbitrario y aproxime el valor apropiado para y.

6.6.2. Matrices

1. Gauss-Jordan

   El algoritmo Gauss-Jordan consiste en una forma de dar solución a un sistema de ecuaciones lineales basado en operaciones fundamentales.

La complejidad computacional de la eliminación gaussiana es aproximadamente n^3. Esto es, el número de operaciones requeridas es n^3 si el tamaño de la matriz es $n \times n$. El algoritmo consiste en los siguientes pasos:[5]

a) Ir a la columna no cero extrema izquierda

b) Si el primer renglón tiene un cero en esta columna, intercambiarlo con otro que no lo tenga

c) Luego, obtener ceros debajo de este elemento delantero, sumando múltiplos adecuados del renglón superior a los renglones debajo de él

d) Cubrir el renglón superior y repetir el proceso anterior con la submatriz restante. Repetir con el resto de los renglones (en este punto la matriz se encuentra en la forma de escalón)

e) Comenzando con el último renglón no cero, avanzar hacia arriba: para cada renglón obtener un 1 delantero e introducir ceros arriba de éste sumando múltiplos correspondientes a los renglones correspondientes

Ejemplo[6]:

$$\begin{cases} 2x & +y & -z & = & 8 \\ -3x & -y & +2z & = & -11 \\ -2x & +y & +2z & = & -3 \end{cases}$$

En nuestro ejemplo, eliminamos x de la segunda ecuación sumando 3/2 veces la primera ecuación a la segunda y después sumamos la primera ecuación a la tercera. El resultado es:

$$\begin{cases} 2x & +y & -z & = & 8 \\ & \frac{1}{2}y & +\frac{1}{2}z & = & 1 \\ & 2y & +z & = & 5 \end{cases}$$

Ahora eliminamos y de la primera ecuación sumando -2 veces la segunda ecuación a la primera, y sumamos -4 veces la segunda ecuación a la tercera para eliminar y.

$$\begin{cases} 2x & & -2z & = & 6 \\ & \frac{1}{2}y & +\frac{1}{2}z & = & 1 \\ & & -z & = & 1 \end{cases}$$

Finalmente eliminamos z de la primera ecuación sumando -2 veces la tercera ecuación a la primera, y sumando 1/2 veces la tercera ecuación a la segunda para eliminar z.

$$\begin{cases} 2x & & & = & 4 \\ & \frac{1}{2}y & & = & \frac{3}{2} \\ & & -z & = & 1 \end{cases}$$

Despejando, podemos ver las soluciones:

[5]Descripción del algoritmo tomado de http://es.wikipedia.org/
[6]Este ejemplo fue tomado de http://es.wikipedia.org/

$$\begin{cases} x & & = & 2 \\ & y & = & 3 \\ & & z & = & -1 \end{cases}$$

Para clarificar los pasos, se trabaja con la matriz aumentada. Podemos ver los 3 pasos en su notación matricial:

Primero:
$\begin{pmatrix} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{pmatrix}$ Después, $\begin{pmatrix} 2 & 0 & 0 & 4 \\ 0 & 1/2 & 0 & 3/2 \\ 0 & 0 & -1 & 1 \end{pmatrix}$

Por último. $\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{pmatrix}$

Si el sistema fuera incompatible, entonces nos encontraríamos con una fila como esta:

$\begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$

Que representa la ecuación: 0x + 0y + 0z = 1, es decir, 0 = 1 que no tiene solución.

2. Exponenciación de matrices (cuadradas)

   Dada una matriz M y un exponente n, la idea de este ejercicio es multiplicar la matriz M por sí misma n veces. ¿Se puede hacer de forma logarítmica con respecto a n?

6.6.3. Listas

1. Ordenar una lista

   Recibe una lista de números y devuelve la lista ordenada. En todo caso busque minimizar el $O(n)$

2. Encontrar una sublista desde la posición i hasta la posición j

   Recibe una lista y dos enteros, el i y el j. Se puede asumir que $i \leq j$.

3. Buscar la cantidad de elementos múltiplos de k en una lista de números

   Recibe una lista de números y un entero. Retorna un entero con la cantidad de números de esa lista que son múltiplos de k.

4. Lista de los primeros n enteros.

   Recibe un número entero y retorna una lista con los primeros n números enteros positivos.
   Ejemplo primerosEnteros(5) =

5. Primeros primos

   Similar al anterior, pero con la salvedad de que se recibe un entero n, pero devuelve los primeros n primos.

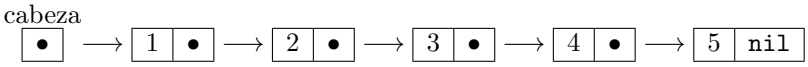


Figura 6.7: Primeros 5 elementos de $\mathbb{Z}^+$

6. Intercambiar dos elementos

   Este tiene como propósito recibir dos enteros y una lista, los enteros son las posiciones e intercambian los elementos que están en esas posiciones.

7. Reemplazar elementos

   Se recibe una lista y dos elementos α y β, de tal manera que reemplaze cada aparición de α por β en la lista.

8. Eliminar duplicados

   Recibe una lista con elementos que pueden estar o no duplicados y devuelve la lista sin esos elementos repetidos.

9. Es Palíndromo

   Un palíndromo es una estructura que se lee igual de derecha a izquierda que de izquierda a derecha. Por ejemplo: *salas*,*ana*,*saira arias*,*oso*,*se van sus naves*.

10. Permutaciones

    Dada una lista indicar todas las permutaciones que esta lista puede tener. Ejemplo: A,B,C ->ABC, ACB, BAC, BCA, CAB, CBA.

    Dada una lista con una permutación particular, hacer un *nextPerm* que permita mostrar la siguiente permutación de esa posición en particular, de modo que si se ejecuta la función $n! - 1$ veces, se obtiene de nuevo la lista original.

6.7. Resumen

En este capítulo se introdujeron los primeros conceptos de conjuntos de datos y cómo manipularlos. En el siguiente capítulo, se verán estructuras con mecanismos de manipulación de datos más particulares, denominados *pilas*, *colas* y *árboles*. En conjunto con las presentes en este capítulo, son una breve introducción a las estructuras de datos más básicas que se deben conocer para aprender a programar correctamente.

Capítulo 7

Pilas, colas y árboles binarios

> Conoces lo que tu vocación pesa en ti. Y si la traicionas, es a ti a quien desfiguras; pero sabes que tu verdad se hará lentamente, porque es nacimiento de árbol y no hallazgo de una fórmula
>
> Antoine de Saint Exupéry

La parte restante del libro está dedicada a aplicaciones o usos de las estructuras mencionadas en capítulos anteriores, entonces más allá de conceptos nuevos, se trata de técnicas, usos o algoritmos particulares sobre los conceptos vistos previamente.

7.1. ¿Qué son las estructuras de datos?

Una estructura de datos es una forma de administrar datos con una política (algoritmo) para la inserción y eliminación de datos. Las hay de dos tipos, las estáticas y las dinámicas, las cuales hacen referencia a la forma en como administran la memoria tal y como se vio en el capítulo 6.

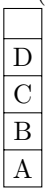
Existen muchas estructuras de datos que se han creado para resolver diferentes problemas, en este capítulo se mencionarán únicamente las estructuras más básicas y generales, tal y como son las pilas, colas y árboles binarios.

7.2. Pilas

Una pila o *stack*[1] es una estructura de datos muy simple. El comportamiento de la pila es muy similar al de una pila de platos, es decir, se colocan (o *apilan*) uno encima de otro y la forma de retirarlos, es tomando el primero, el que está

[1] Nombre en inglés

más arriba y por ende, el último que entró. Se dice que el comportamiento de las pilas es UEPS (Último en Entrar, Primero en Salir) o más popularmente LIFO (Last In, First Out). Ejemplo:



En esa figura, el último en ser ingresado a la pila fue 'D' y el primero en haber sido ingresado fue 'A'. Más puntualmente, el orden en que se ingresaron los elementos a dicha pila fue: {A,B,C,D}.

7.2.1. Partes de la pila

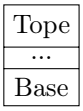
La pila tradicionalmente tiene dos partes importantes:

1. Tope:

   El tope es el próximo elemento en ser eliminado de la pila.

2. Base:

   La base es el primer elemento ingresado en el momento en que la pila está vacía, es decir, se trata del elemento de mayor antigüedad en la pila.



Las operaciones de insertar y eliminar también tienen un nombre particular en las pilas, insertar se denomina **push** y al eliminar se le denomina **pop**. En el fondo, el tener una pila en su forma más pura consiste en tener algún mecanismo de administración de datos que posea al menos estas dos operaciones.

7.2.2. Pilas con memoria estática

Una pila implementada con memoria estática es principalmente un sector de memoria donde se debe guardar dos direcciones, la de la base y la del tope. Dado que se trata de un segmento de memoria lógico y secuencial, no es necesario almacenar ningún otro tipo de información (pero en algunos lenguajes se debe reservar el sector de memoria, con su tamaño inicial).

La principal desventaja es que se tiene un espacio de memoria limitado y por ende, una cantidad máxima de elementos que deben de ponerse en la pila.

Inserción y eliminación en una pila estática

Dado que una pila estática se cuenta con un valor máximo de elementos que pueden ser ingresados en ella, el algoritmo debe de ser como se muestra a continuación:

```
push( atributo )
    si(tope < maximo)
        pila[tope] <- atributo
        tope <- tope+1

pop ()
    si (tope > 0)
        tope <- tope-1
        retornar pila[tope+1]
```

Es destacable que en ambos casos se trata de algoritmos que son $O(1)$

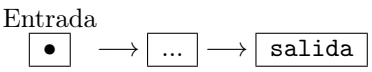
7.2.3. Pilas con memoria dinámica

Una pila con memoria dinámica es en realidad una estructura equivalente a una lista, con la particularidad de que sólo se pueden hacer las operaciones de *push* y *pop*. Se cuenta con la ventaja que pueden crecer tanto como la memoria lo permita, pero siempre es importante que cada nodo recuerde la dirección de su sucesor, al menos.

Por razones de simpleza en los algoritmos es probable que sea más cómodo utilizar el primer elemento como tope, de manera que un *push* sea un insertar al inicio y un *pop* eliminar al inicio. Claro que es importante hacer la validación de que no se encuentre vacía a la hora de eliminar.

7.3. Colas

Una cola o *queue*[2] es una estructura de datos que se asemeja más a la forma tradicional en cómo se hace cola o fila en algunos lugares, por ejemplo un banco donde las personas entran y en el mismo orden en que llegaron, son atendidos. El comportamiento de una cola es PEPS (Primero en Entrar, Primero en salir) o más popularmente sus siglas en inglés FIFO (First-In, First-Out). Un ejemplo gráfico de esto se puede apreciar en la figura siguiente.



7.3.1. Partes de la cola

La cola tradicionalmente tiene dos partes importantes:

1. Frente:

   El frente es el próximo elemento en ser tomado de la cola, es el lugar por donde salen los elementos.

[2] ídem

2. Fin:

   El fin es por donde se ingresan los elementos, cada nuevo elemento ingresado está en el final de la cola.

7.3.2. Colas con memoria estática

En principio es altamente deseable que tanto el insertar como el eliminar tenga un $O(1)$ y esto es sencillo, pero es importante recordar un poco de teoría de números para poder garantizarlo.

Como se ha indicado, a una cola se le inserta por un lado y se saca por otro lado, pero sería idóneo tener sólo dos posiciones en un arreglo (segmento estático de memoria), donde se puede operar las posiciones.

Supongamos que se cuenta con un segmento de tamaño t, los algoritmos serían tal y como se presentan a continuación:

```
encolar( elemento )
    si (posFin = -1)
        posFin <- 0
        posFrente <- 0
        cola[0] <- elemento
    en otro caso si (posFin != posFrente)
        cola[posFin] <- elemento
        posFin <- posFin -1
        posFin <- (posFin) mod t
        si(posFin = posFrente)
            posFin <- -1
            posFrente <- -1

desencolar ()
    si (posFrente != -1)
        elemento-de-retorno <- cola[posFrente]
        posFrente <- posFrente + 1
        posFrente <- posFrente mod t
        retorna elemento-de-retorno
```

7.3.3. Colas con memoria dinámica

En el caso de las colas implementadas con memoria dinámica, una vez más, es representada en una lista con la diferencia que sólo se añade por el frente y se inserta por el fin.

Para lograr que el algoritmo sea $O(1)$ es importante sacrificar un poco de espacio en memoria y guardar dos direcciones una para el primer nodo y otra para el último nodo. El fin estará en el este último nodo y el frente, como es natural estará en el primero.

El algoritmo para la inserción y eliminación sería como se indica:

```
encolar ( elemento )
    si(primero = null)
        primero <- nodo(elemento)
        ultimo <- primero

    en otro caso
        ultimo.sgte <- nodo(elemento)
        ultimo <- ultimo.sgte


desencolar ()
    si(primero != null)
        valor <- primero.elemento
        primero <- primero.sgte
        retornar valor
```

7.4. Árboles

Hasta el momento se han mostrado estructuras de datos que son lineales, es decir listas en sus diferentes formas, tanto en pilas como en colas.

Los árboles deben su nombre a la forma gráfica que proyecta, pues tiene cierta similitud con un árbol.

Sin embargo los árboles son estructuras de datos más complejas que a la vez proveen ciertas ventajas para operar datos, con la complicación de tener algoritmos que implican mayor cuidado para poder insertar, eliminar o buscar elementos.

7.4.1. Partes de los árboles

Al igual que las listas, los árboles están compuestos por nodos, para facilitar su estudio se le han dado nombres a los nodos dependiendo de su lugar en el árbol. En la figura 7.1 de la página 112, se pueden apreciar por su nombre algunos de los nodos. Con base en dicha figura se numeran las siguientes partes:

1. *Raíz*

   Todo nodo que no tiene padre, desde el cual "descienden" los otros nodos, es considerado la raíz. En la figura la raíz es r.

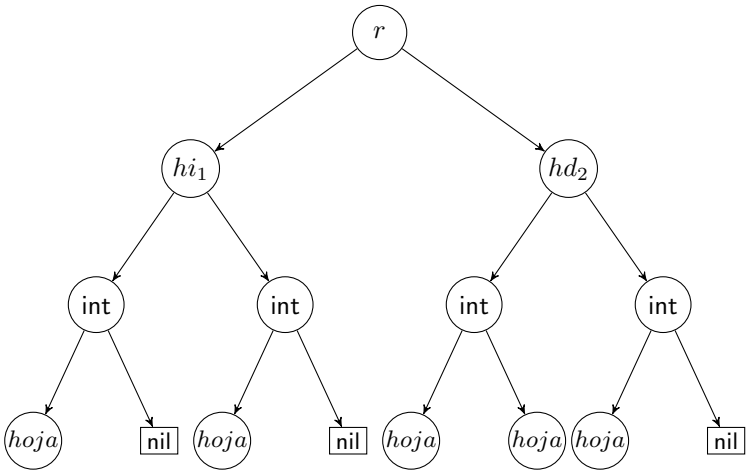


Figura 7.1: Árbol binario

2. *Hijo y padre*

   Esta posición es relativa, en el caso de los árboles binarios, se tiene dos hijos, un *hi* que es el hijo izquierdo y un *hd* que es el hijo derecho. Naturalmente, el padre es quien señala a los dos hijos.

3. *Hoja*

   Una hoja es un nodo que no tiene hijos (ninguna cantidad de hijos)

4. *Interior*

   Un nodo interior, se define como todo nodo que no sea ni raíz, ni hoja.

7.4.2. Conceptos asociados a árboles

A continuación se definen algunos de los atributos principales de los árboles que son de mucha utilidad para hacer cálculos y operaciones con el árbol.

1. Altura

   Esta se define como la cantidad de aristas que separan a la raíz del nodo más alejado de esta

2. Nivel

   Estos son los nodos que distan a la misma distancia (número de aristas) de la raíz.

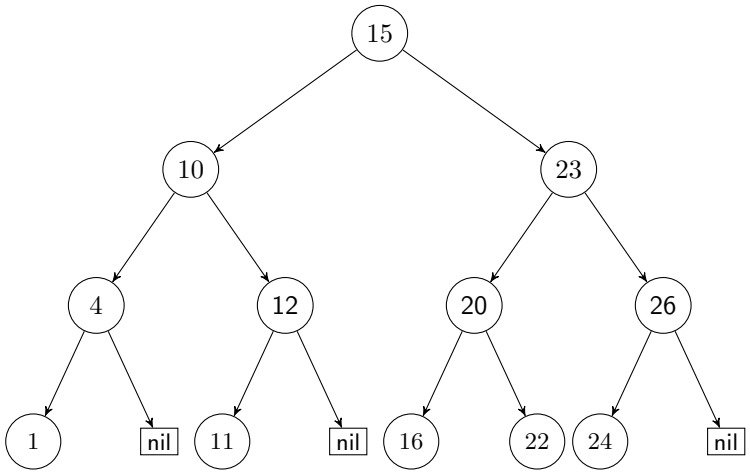


Figura 7.2: Árbol binario de búsqueda

3. Amplitud

   La amplitud de árbol es el número de nodos que tiene el nivel que cuenta con la máxima cantidad de nodos de todo el árbol.

4. Subárbol

   Es el árbol que tiene como raíz a un nodo que es hijo de otro nodo padre que es parte de un árbol mayor. En realidad, puede observarse que cada nodo de un árbol es en sí mismo raíz de un subárbol.

5. Grado

   El grado de un árbol hace referencia a la cantidad de hijos que tiene el nodo que más hijos tiene.

7.4.3. Árboles binarios

Un árbol binario es un árbol, con la particularidad que los nodos tienen a lo sumo, dos hijos, por ende es un árbol de grado 2.

7.4.4. Recorridos en un árbol binario

Por su naturaleza hay tres recorridos que se pueden realizar, cambiando el orden en como se muestra la raíz. Para mostrar los recorridos se utilizará la figura 7.2 de la página 113.

1. Preorden: **Raíz Izquierdo Derecho**

   15-10-4-1-12-11-23-20-16-22-26-24

2. Inorden: **Izquierdo Raíz Derecho**
   1-4-10-11-12-15-16-20-22-23-24-26
3. Postorden: **Izquierdo Derecho Raíz**
   1-4-11-12-10-16-22-20-24-26-23-15

Los algoritmos para encontrar los recorridos son naturalmente recursivos:

```
orden(Nodo n)
    si n no es nulo
        **********Pre
        orden(n.izq)
        **********In
        orden(n.der)
        **********Post
```

En principio cada uno de los asteriscos debe reemplazarse por la función de mostrar o incorporar en una lista los elementos del nodo n, tomando como base el tipo de orden, es decir si es el preorden, se reemplaza el Pre por mostrar el atributo del elemento n.

7.4.5. Árboles binarios de búsqueda

Un árbol binario de búsqueda es un caso particular de árbol binario, se trata de un árbol que la forma de organizarse es poner los mayores a la raíz, de un lado de la raíz y los menores, del otro lado. De manera indistinta puede ser mayores a la derecha y menores a la izquierda o viceversa, sin embargo, de manera estándar es tal y como se muestra en la figura 7.2 de la página 113.

Inserción y eliminación

Inserción

El insertar es un proceso muy sencillo, se trata de un método recursivo el cual agrega un dato de una forma así:

```
insertar( dato , nodo)
    si (nodo está vacío)
         nodo <- dato
    si (nodo.valor < dato)
         insertar (dato, nodo.HijoDerecho)
    en otro caso
         insertar (dato, nodo.HijoIzquierdo)
```

Como se puede apreciar, se recorre (desde la raíz) el árbol a través de la ruta según corresponda, si es mayor, a la derecha y si es menor a la izquierda. La idea es que ese recorrido sea único de modo que si hay que buscar un elemento, se pueda hallar su posición usando la misma regla.

Eliminación

El eliminar es un proceso un poco más complejo y se puede dividir en tres casos:

1. Eliminar una hoja

   Este es el caso más sencillo de todos, simplemente se elimina

2. Eliminar un nodo que tiene sólo un hijo

   En este caso, el hijo llega a ocupar el lugar del padre[3]

3. Eliminar un nodo con dos hijos

   Este es el caso más complejo de todos, se trata de un problema que primero tiene que buscar el mayor de los menores o el menor de los mayores (que siempre será una hoja), para que ocupe el lugar del nodo eliminado, esto permite reemplazar al nodo eliminado sin tener que perder que los mayores, seguirán siendo mayores que el sustituto y los menores van a ser menores que el sustituto, también. Nótese que en el fondo, se tiene que buscar el nodo más a la derecha de los nodos izquierdos o el más a la izquierda de los derechos.

7.5. Grafos

Suponga que se tiene un conjunto de elementos, a los que se denominan *nodos* y sobre ellos se establece una relación (no reflexiva)[4], a esto se le denomina grafo.

Las estructuras que se han descrito en este capítulo son todos casos particulares de grafo, por ejemplo, una lista, una cola y una pila, son en realidad, nodos relacionados unos con otros mediante *aristas* cuyo valor es precisamente el par de nodos que están relacionados.

Sobre los grafos existen algunos conceptos (que se pueden generalizar a las demás estructuras de datos):

1. Camino

   Un camino es una secuencia de nodos de tal manera que para dos nodos seguidos de la secuencia n_i, n_{i+1}, existe la arista $(n_i, n_{i+1}$.

2. Grafo dirigido

   Es aquel donde las relaciones de las aristas no son simétricas (de manera análoga, un grafo no dirigido, es aquel donde las relaciones son simétricas).

3. Grafo conexo

   Un grafo es conexo si para todo par de nodos, existe un camino entre ellos

4. Grafo completo

   Un grafo es completo si todas las posibles relaciones están presentes, es decir, todos los nodos están conectados con cada uno de los nodos restantes.

[3] Recordemos que se debe respetar lo de menores a un lado y mayores al otro

[4] Tal y como se define en la sección 1.3 de la página 21

7.6. Problemas

7.6.1. Pilas y colas

1. Ordenar elementos en una pila

   Utilizando únicamente operaciones de pila (push y pop) y a lo sumo una pila auxiliar, ordene de menor a mayor los elementos de una pila. Eventualmente se permite hacer push de toda una pila de forma que se enlazen las pilas.

2. Utilice una pila para implementar torres de hanoi **sin** utilizar recursión

3. Realice un programa que encuentre un problema para las 8Reinas.

4. Realice un programa que usando backtracking pueda hacer el recorrido del caballo de ajedrez en un tablero de $n \times m$ y una posición inicial i, j. El recorrido del caballo consiste en cubrir todo el tablero sin repetir escaques.

7.6.2. Árboles

1. Altura

   Escriba un programa que determine la altura de un árbol

2. Orden

   Escriba un programa que determine el preorden de un árbol

   Escriba un programa que determine el inorden de un árbol

   Escriba un programa que determine el postorden de un árbol

3. Niveles

   Escriba un programa que dado un valor, devuelva todos los niveles en que aparece dicho valor en un árbol binario arbitrario

   Escriba un programa que reciba un nivel y que devuelva todos los nodos de ese nivel en un árbol binario arbitrario

   Escriba un programa que busque el nivel más ancho del árbol (el que tiene más nodos) y lo muestre.

4. Avanzados

   Escriba un programa que dado un árbol indique si se trata de un árbol balanceado o no.

   Escriba un programa que reciba un número N y genere todos los árboles balanceados de N nodos.

   Escriba un programa que dado una altura H, genere todos los árboles balanceados de altura H.

   Escriba un programa que dado el inorden y el preorden del mismo árbol binario, reconstruya el árbol original.

7.7. Resumen

En este capítulo 7 se definieron los conceptos de pila, cola y árboles, finalmente se enlazaron con el concepto genérico de grafo y se definieron las partes de este. También se describieron algoritmos sencillos para realizar operaciones con estas estructuras. Se presentaron además, varios problemas cuya realización puede ser significativa para la comprensión de los temas desarrollados en este capítulo. El capítulo 8 contiene varias curiosidades de problemas que se pueden resolver al darle un uso particular a los conocimientos adquiridos en este libro. Se busca que no sean particularmente difíciles, pero sí interesantes y representativos. Se espera que resulten divertidos de desarrollar para los lectores.

Capítulo 8

Problemas clásicos de programación

Muchas cosas se reputan imposibles antes de haberse realizado

Plinio el Viejo

La mayoría de los buenos programadores programan, no porque esperan que se les pague o por adulación por parte del público, sino porque es divertido programar

Linus Torvalds

8.1. Introducción

Este capítulo intenta mostrar problemas clásicos de programación, conocerlos es parte del bagaje cultural de cualquier programador. Algunos plantean soluciones sencillas a problemas que en principio parecían complejos, otros son ejemplos de como se han modelado y solucionado otros problemas que han sido tratados como NP (cuya solución es tradicionalmente por *backtracking*).

8.2. Cálculo de π

Este quizá sea el menos "avanzado" de los problemas planteados en este capítulo porque a pesar de parecer algo complicado, calcular pi es realmente sencillo. En esta sección, se explorarán varias maneras de programar el cálculo de pi. Una de ellas será mediante un algoritmo probabilístico basado en simulación.

8.2.1. Series de Taylor

Una de las formas más conocidas y precisas para calcular π es mediante una suma infinita. Si se superara el problema de la precisón flotante y se manejan los decimales con cuidado, se satisface la siguiente ecuación:

$$\pi = 4 \times \sum_{i=0}^{\infty} \frac{-1^i}{2i+1}$$

Cuantos más sumandos y cuánto más preciso sea la precisión, más preciso será el resultado.

8.2.2. Cálculo de pi por simulación

En el capítulo 3 se mencionó que hay dos tipos de algoritmos, los determinísticos y los probabilísticos. Hasta este punto en el libro sólo se han desarrollado algoritmos determinísticos, sin embargo, existen un gran número de algoritmos probabilísticos muy útiles que se utilizan principalmente para dar una solución suficientemente buena[1] a problemas cuya complejidad es elevada.

Bases de geometría

Para encontrar el valor de π por simulación (de forma probabilística) primero debemos de ver una figura como la 8.1:

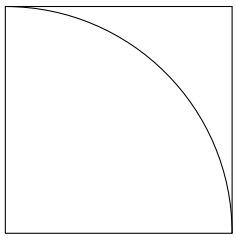


Figura 8.1: Cuadrado con sector circular inscrito

Supongamos que el lado del cuadrado mide $1u$ de forma arbitraria, por ende el área del cuadrado es $(1u)^2$, por otro lado, el sector circular, es una cuarta

[1] Esto porque la solución óptima es muy difícil de obtener, normalmente se trata de problemas NP, tal y como se muestra en el capítulo 5

parte del círculo completo. Como se puede observar ese círculo tendría también como radio $1u$, por lo que el área del círculo mostrado es $(\frac{\pi}{4}u)^2$[2], por ende el área del sector circular dividido por el área del cuadrado, es $\frac{\pi}{4}$.

Considerando que un área es en realidad los puntos de un plano, si lanzo n puntos aleatorios dentro del cuadrado y cuento la cantidad de puntos d que caen dentro del círculo, la razón $\frac{d}{n}$ sería equivalente a $\frac{\pi}{4}$, siendo mucho más exacto cuánto mayor sea n, es decir, la cantidad de puntos.

Para garantizar que un punto aleatorio esté dentro del cuadrado, se necesita que haya un aleatorio entre 0 y 1 (porque la esquina superior del cuadrado, sería el punto $(1, 1)$, mientras que la esquina inferior sería $(0, 0)$ por ende cualquier punto (x, y) estaría dentro del cuadrado sii $0 \leq x, y \leq 1$.

Distancia entre puntos

El siguiente tema es como calcular si un punto está dentro de la circunferencia. Para ello nos apoyaremos en el teorema de Pitágoras que nos sirve de manera general a calcular la distancia entre dos puntos.

Veamos el siguiente plano cartesiano en la figura 8.2

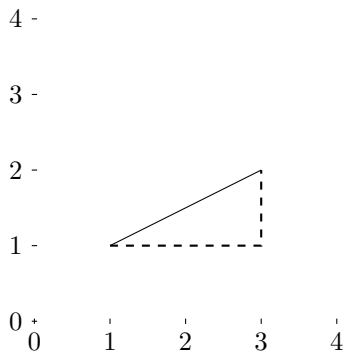


Figura 8.2: Segmento de recta entre (1,1) y (3,2)

Como se puede ver entre dos puntos cualesquiera de un plano cartesiano, se forma una hipotenusa de un triángulo rectángulo, cuyos catetos ($c_1 y c_2$) son la diferencia entre las x y las y de cada punto. Por consiguiente, se cumple la ecuación $h^2 = c_1^2 + c_2^2$.

Entonces la distancia entre dos puntos (x_1, y_1) y (x_2, y_2) es:

$$S((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Calcular π

Para el caso particular que nos atañe, calcular π se trata de saber si un punto está dentro del sector circular o no, para ello es indispensable calcular la

[2]Es importante tomar en cuenta que el área del círculo es πr^2 y en este caso, $r = 1$

distancia entre un punto cualquiera y el origen, que es $(0, 0)$, donde estar dentro del círculo implica estar en una distancia menor o igual a 1.

8.3. El juego de la vida

Este se trata de un juego matemático que fue propuesto por el matemático John Horton Conway en 1970.

Este juego, en realidad, no tiene jugadores, sino que sólo se coloca una posición inicial en una matriz y se observa su evolución en el tiempo, generación a generación. En la matriz inicial, cada casilla de la matriz se le denomina *célula*[3] y tiene dos estados, 1 (viva) o 0 (muerta).

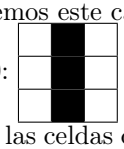
Las reglas son simples:

1. Si una célula viva, tiene 2 o 3 casillas vivas alrededor (incluyendo diagonales), entonces permanece viva, si no, muere (soledad o sobrepoblación)

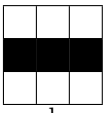
2. Si una casilla muerta tiene exactamente 3 vivas alrededor, entonces nace

Lo interesante de este juego es que dada una posición inicial, este evoluciona dependiendo de la configuración inicial.

Consideremos este caso:

Generación 0: 

En este caso, las celdas de los extremos tienen sólo una celda viva al lado, por lo que en la siguiente generación mueren. La celda del centro, tiene 2 celdas vivas al lado, por lo que se mantiene viva la otra generación. Por lo que queda en la siguiente generación de esta forma:

Generación 1: 

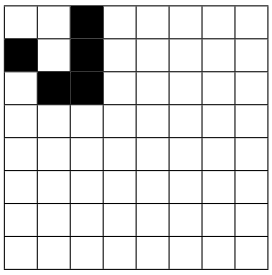
De forma análoga, la generación 1, produce un estado equivalente a la generación 0.

Lo realmente interesante es que a partir de diferentes estados iniciales se pueden hacer generaciones que tienen comportamientos muy interesantes, algunas convergen a ninguna célula viva y hay otras en las siempre crecen en número de células vivas (si se pudiera poner la matriz infinito).

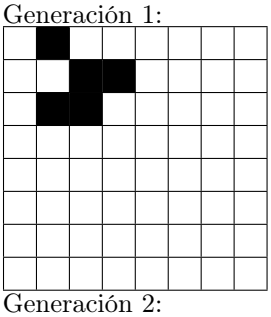
Por ejemplo, la siguiente es una *"nave espacial"* y se denomina *glider*

Generación 0:

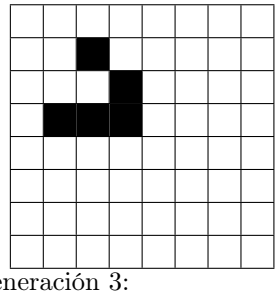
[3] Que es la unidad de la vida, por eso, el juego de la vida



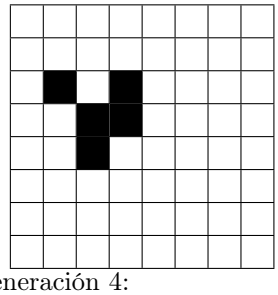
Generación 1:



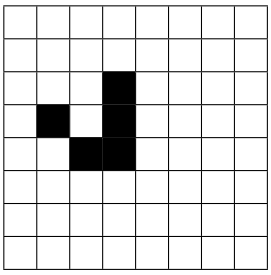
Generación 2:



Generación 3:



Generación 4:



Como se puede ver la generación 4 es igual a la generación 0, sólo que proyectada un escaque a la derecha y 2 hacia abajo. Si se continúa sobre una matriz más grande, la *nave* se mueve hacia la esquina inferior derecha. El ciclo se repetiría corriéndose siempre hacia abajo y a la derecha.

8.4. Generación de números aleatorios

Uno de los problemas más interesantes en la creación de algoritmos probabilísticos, como se puede ver, en la sección 8.2.2 de la página 121, es necesario crear números aleatorios. Esto quiere decir que sean números que no sigan un patrón específico y que puedan cubrir un conjunto de números en su totalidad.

El problema principal de esto, radica en que una computadora como se vio en el capítulo 2 la computadora es una máquina de estados determinística, por lo que puede hacer es ejecutar algoritmos. Sin embargo, eso se contrapone al deseo de tener valores aleatorio.

Para ello se utilizan en realidad funciones (determinísticas) que simulan generar valores aleatorios. Tienen un ámbito de llegada, el cual se vuelve cíclico luego de ciertas corridas, esta función debe parecer ser aleatoria y requiere un valor inicial para ser usado como ***semilla***.

La secuencia de valores va a estar dada en función de esa semilla inicial, por ello, la semilla debe ser muy cuidadosamente seleccionada.

A esta *semilla* se le denomina s_0 y $f(s_0) = s1$, luego $f(s_1) = s_2$ y de manera genérica $f(s_n) = s_{n+1}$

8.4.1. Método congruencial mixto

Para explicar este método, se debe tener un ámbito, para este ejemplo, se utilizará un codominio: {0,1,2,3,4,5,6,7,8,9}.

Suponga la función:

$$f(n) = (11n + 7)(10)$$

Si utilizamos la semilla inicial $s_0 = 7$, se obtiene la siguiente secuencia:
$f(7) = 4$ $f(4) = 1$ $f(1) = 8$ $f(8) = 5$ $f(5) = 2$ $f(2) = 9$ $f(9) = 6$ $f(6) = 3$ $f(3) = 0$ $f(0) = 7$
De esta manera, se puede apreciar que se generan los números del ámbito, de forma *"aleatoria"*. El método congruencial mixto, requiere de tres valores, el coeficiente, el corrimiento y el módulo (que es lo que da el nombre al método),

en este caso se utilizaron los valores 11, 7 y 10. Es importante notar que no cualquier conjunto de valores puede hacer que se cubra el ámbito completo.

Por ejemplo, si se tiene $f(n) = (4n + 3)(10)$, si se usa la semilla $s_0 = 3$, la secuencia sería $3, 6, 7, 5, 3$, que como se puede ver, el 3 se repite reiniciando la secuencia tan sólo 4 números después.

8.4.2. Cuadrados medios

La idea de los cuadrados medios consiste en tener un número de la forma $a_1a_2a_3(...)a_n$, como semilla s_0 y la semilla s_1 sería los n dígitos medios de $(s_0)^2$.

Por ejemplo, si tengo $n = 4$ y sea $s_0 = 4248$, $(s_0)^2 = 18045504$ y los 4 dígitos de en medio son: $(18 - 0455 - 04)$ 0455, de modo que $s_1 = 455$. Nuevamente $(s_1)^2 = 207025$, de esta manera $s_2 = 702$ y si seguimos el mismo método se obtiene $s_3 = 9280$, $s_4 = 1184$, $s_5 = 4018$, $s_6 = 1443$, $s_7 = 822$. Se puede ver la idea de que cada número parece *aleatorio* con respecto al anterior. Sin embargo, se puede demostrar que no se cubre el ámbito completo de los números de n dígitos, por lo que este método no es perfecto. Además, depende mucho de la semilla correcta para generar los valores que se necesitan.

8.4.3. Obtención de números aleatorios perfectos

Un modelo que permite la obtención de un valor aleatorio perfecto, debe depender completamente del azar, por lo que existe un modelo de *computación cuántica*, el cual se basa en ubicar la posición de un electrón en un instante dado en un átomo de hidrógeno al vacío. El electrón está en movimiento continuo dependiendo de la temperatura, así se marca su velocidad.

Para saber la ubicación del electrón en un momento dado, se necesita de un fotón (partícula de luz) para que *choque* con el electrón y su *rebote* nos indica donde se encontraba el electrón en el momento en que fue impactado por el fotón. Sin embargo, esta posición es aleatoria, por lo que si mido esa posición en un momento arbitrario, el valor que obtenga asociado en una función cuyo ámbito es $\{0, 1\}$, y se miden n veces, puedo tener un número binario de n dígitos, completamente arbitrario.

Éste método es completamente aleatorio y está siendo utilizado por algunos bancos para generar los pines de las tarjetas de crédito, de modo que no se pueda predecir el pin de un tarjetahabiente.

Actualmente hay varios sitios en internet que realizan algoritmos similares a los descritos en esta sección, con diferentes fenómenos físicos (campo electromagnético, computación cuántica, etc) para poder brindar números aleatorios perfectos, que estén al margen de una implementación algorítmica.

8.5. Fractales

Un fractal es una figura que se repite en sí misma varias veces de forma recursiva. En la figura 8.3[4] podemos apreciar la curva de Hilbert. Podemos ver como se replica el mismo cambio en cada una de las líneas sobre cada iteración. En la figura 8.3[5] podemos apreciar diferentes fractales.

Lo interesante de los fractales es que resultan muy sencillos de programar, con una función recursiva, que escala y rota la imagen anterior varias veces sobre sí misma.

8.6. Compresión de archivos

La primera forma en como se logró comprimir archivos de forma efectiva fue mediante los árboles de Huffman, llamados así por su creador David Huffman.

El algoritmo está basado en el cálculo de frecuencias de los bytes que aparecen en el archivo. Es decir, se debe contar cuántas veces aparece el valor de cada byte de un archivo. Luego construye un árbol binario que busque balancear dichas frecuencias. Esto se logra tomando los dos nodos (sin padre), de menor frecuencia y poniéndolos con un padre que tiene la frecuencia de ambas (su suma) para así seguir el proceso y tomar los dos nodos (sin padre) de menor frecuencia y se repite el procedimiento hasta que se cuente con un árbol único.

Una vez obtenido el árbol, cada hoja de éste será conformada por uno de los bytes y su respectiva frecuencia, y los nodos interiores serán padres con la suma de las frecuencias de sus hijos inmediatos.

La característica de los árboles es que a partir de su nodo fuente (la raíz), se cuenta con una ruta única para cada uno de los nodos, lo que también ocurre con las hojas (bytes del archivo original) y esta ruta será más corta cuanto más cerca de la raíz se encuentra, es decir, será más corta cuanto más frecuente sea.

Una vez halladas las rutas para cada hoja, se procede a asignarles un 1 para izquierda y un 0 para derecha[6], y se genera una tabla de códigos binarios **únicos** para cada valor del archivo. Finalmente se debe reescribir el archivo, sólo que reemplazando la aparición de cada byte por su código (probablemente este código resulte de menor longitud de un byte, pero si fuera mayor, habría otros más frecuentes, más cortos), lo que hace el archivo comprimido más pequeño o igual en el peor de los casos. Sólo se necesita guardar la tabla de códigos para poder descomprimir el archivo, cuyo proceso es completamente inverso al anterior, cada código es reemplazado por su byte original.

[4] Imagen creada en LaTeX por Marc van Dongen

[5] Imágenes creadas en LaTeX por Stefan Kottwitz y Andrew Stacey

[6] Puede ser al revés sin ningún problema

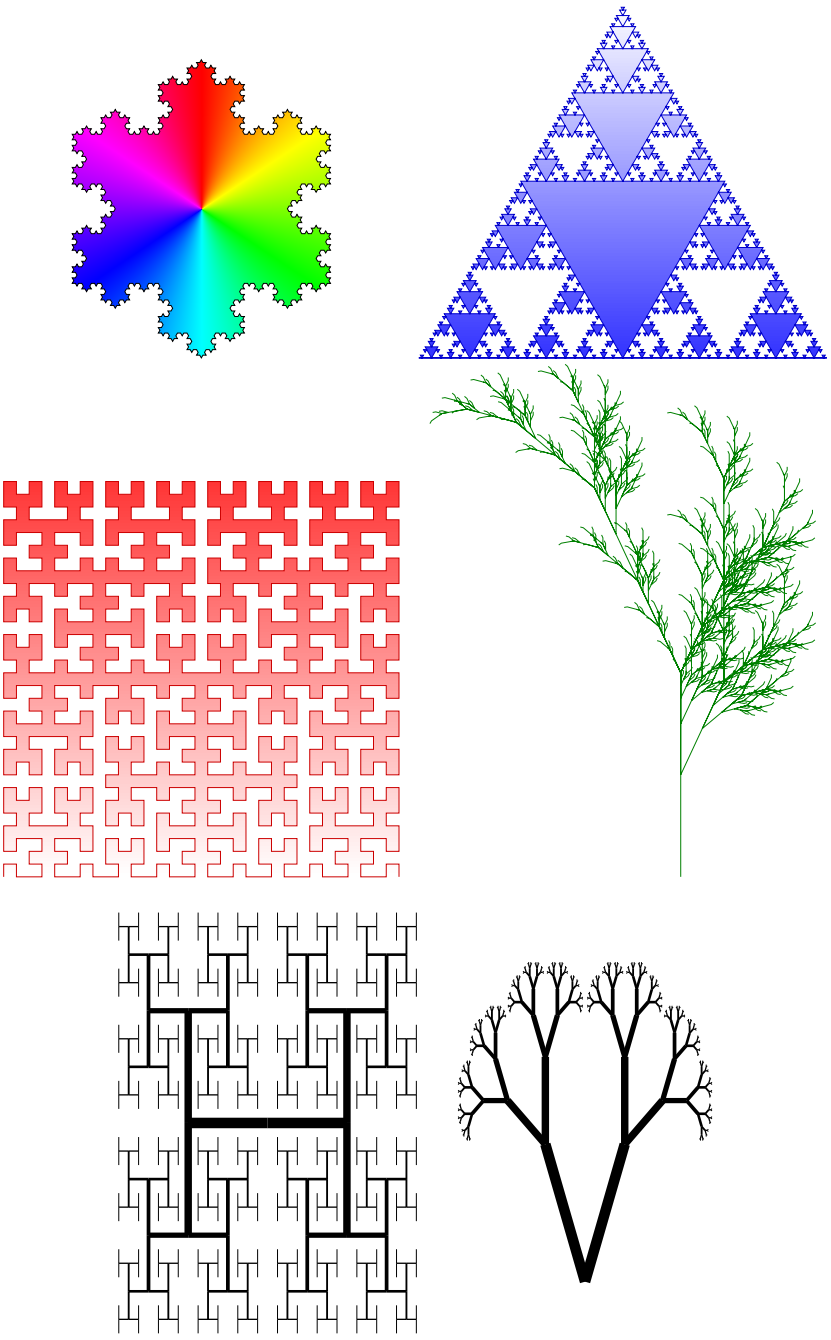


Figura 8.3: Otros fractales

8.7. Uso de la pila: *Backtracking*

El *backtracking* es una de las técnicas de programación más sencillas y capaz de resolver casi todo tipo de problemas de búsqueda y control [7] o de problemas de satisfacción de restricciones[8], con la salvedad de que no siempre lo logra de la manera más eficiente.

Un *backtracking* es tradicionalmente recursivo, dejando que la pila del sistema sea la responsable de ejecutarlo, sin embargo, se puede hacer una pila de forma manual para poder hacerlo de forma manual.

El *backtracking* recibe su nombre porque hace retroceso de ciertas posiciones cada vez que encuentra un camino sin solución.

8.7.1. El problema de las NReinas

Para comprender el problema de las NReinas apropiadamente es necesario primero comprender la forma en como una reina se mueve en un tablero de ajedrez. La imagen 8.4 muestra como es que una reina puede moverse en el tablero. El problema de las n-reinas consiste en colocar en un tablero de $n \times n$ n reinas de modo que no se ataquen entre ellas.

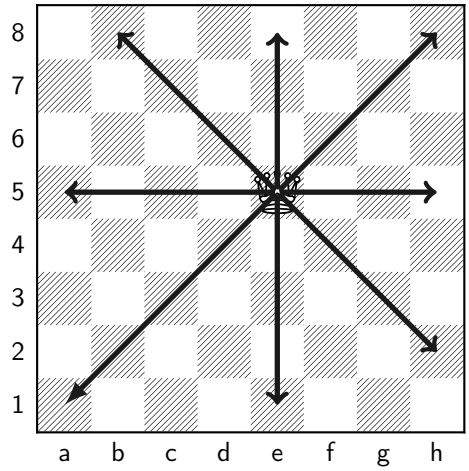


Figura 8.4: Movimientos de la Dama

[7] Cómo encontrar 4 cuadrados que sumen un número o una ruta de un caballo de ajedrez en un tablero de modo que recorra todo el tablero sin repetir casilla o colocar n reinas en un tablero cuadrado de $n \times n$ casillas sin que ninguna reina ataque a otra

[8] Como un laberinto, un sudoku o un kakuro

8-Reinas

En la figura 8.5 de la página 129 se puede ver una posible solución para el problema de las 8 reinas.

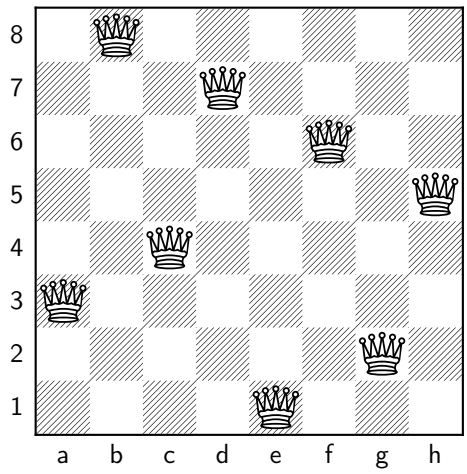


Figura 8.5: Solución para las 8Reinas

8.7.2. Bactracking con la pila

La idea consiste en recorrer todo un árbol, como el que se muestra en la figura 8.6 en la página 130, donde se debe encontrar las soluciones que se pueden ver en las hojas que están al centro de la imagen, aquellas cuyo nivel es cuatro. Es importante considerar que vamos a recorrer esta estructura pero utilizando únicamente una pila.

La idea sería poner la raíz del árbol en la pila y comenzar con la siguiente secuencia de pasos.

```
mientras(pila no vacía)
    Tope = pila->sacar
    if(Tope es solución) //Acción de solución
    else
        for (hijo en Hijos de Tope)
            pila->meter(hijo)
```

Tomar en cuenta que la acción de solución puede variar si se quiere ***una*** sola solución, se quieren ***algunas*** soluciones o si se quieren ***todas*** las soluciones.

- Una solución

  Se retorna la solución y listo

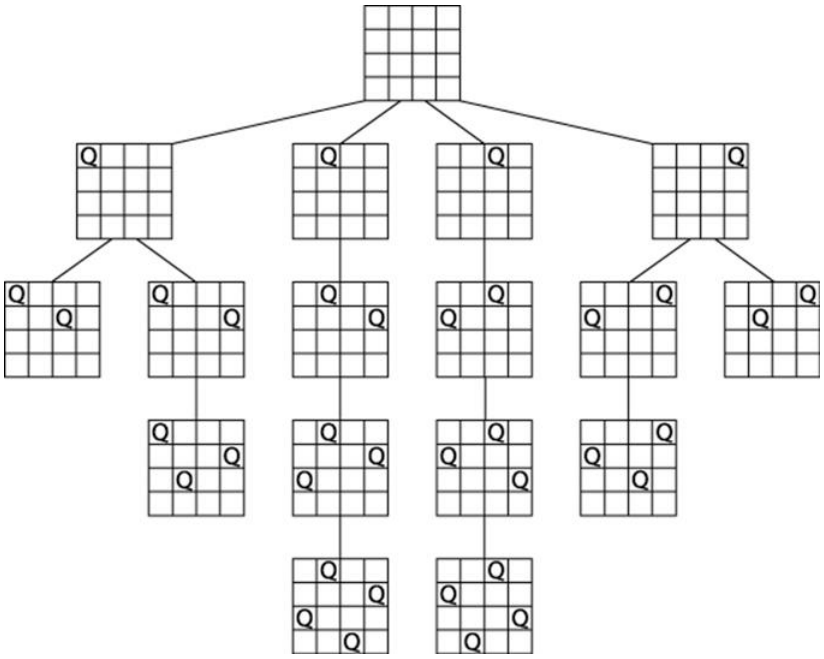


Figura 8.6: Recorrido completo de *Backtracking* para 4 reinas

- Todas las soluciones

  Cada vez que se encuentre una solución se acumula en una lista

- Algunas soluciones

  Dado que esto puede implicar que se deseen algunas soluciones un poco aleatorias, bastaría con barajar la pila cada vez que se encuentra una solución, antes o después de ingresarla a la lista.

8.8. Problemas

8.8.1. Cálculo de π

Una característica de los algoritmos probabilísticos sobre los determinísticos es que tienden a aproximarse más rápido a una solución que los determinísticos equivalentes, sin embargo, al aumentar el número de ejecuciones, los probabilísticos tienden a estancarse en óptimos locales, mientras que los determinísticos se aproximan a la solución.

Para ejemplificar lo anterior, el ejercicio propuesto es el siguiente:

1. Programar un método que calcule π por el método de simulación descrito en 8.2 de la página 120.

2. Calcular π por la serie de Taylor descrita en la misma sección

3. Hacer un programa que para cada punto creado y para cada sumando creado muestre en pantalla los valores que llevan calculados de π, de modo que se note como con pocos puntos, el programa probabilístico, se acerca más al valor π correcto, pero al aumentar el número de sumandos, éste se acerca precisamente al valor correcto.

8.8.2. El juego de la vida

Averigüe como programar una matriz (o *grid*) con interfaz gráfica en el lenguaje de su preferencia. Esta matriz debe ser de $n \times m$. Cada celda va a tener dos colores diferentes, verde si está muerta y rojo si está viva[9], y va a actualizarse periódicamente la matriz para mostrar la evolución del juego de la vida.

La configuración inicial, podría ser leída de archivo o pasada como una matriz de valores booleanos.

8.8.3. Fractales

Se recomienda avergiuar para el lenguaje particular como se hace para dibujar una línea entre dos puntos (inicialmente se puede considerar que la línea es recta, luego se generaliza para curvas). Una vez con esto claro, se sugiere programar los siguientes fractales:

1. Curva de Koch

2. Curva de Levy

3. Curva del Dragón

4. Curva de Hilberth

8.8.4. Backtracking

Para ejercitar el tema de backtracking, se sugieren estos programas:

1. Sudoku (Un programa que reciba un sudoku con ceros en las posiciones vacías y lo rellene respetando las reglas del sudoku).

2. El recorrido del caballo de ajedrez. (Dada una posición i, j en una matriz $n \times m$, recorrer todo el tablero sin repetir casillas con un caballo de ajedrez – con las jugadas legales). El programa deberá devolver la lista de casillas recorridas.

[9] Los colores pueden ser otros de acuerdo con las preferencias del programador

8.9. Resumen

Este capítulo 8 trató sobre una serie de aplicaciones prácticas a diferentes conocimientos expuestos en el libro, esto varía desde sucesiones (cálculo de π) hasta árboles (compresión de archivos), pasando por matrices (juego de la vida). En particular cada uno de las secciones se puede programar, aunque no esté en la sección de problemas, produciendo resultados muy divertidos y que pueden entretener a los programadores que, como dijera Linus Torvalds, *se divierten* al programar.

Apéndice A

Sobre paradigmas de programación

> Los paradigmas son poderosos porque crean los cristales o las lentes a través de los cuales vemos el mundo. El poder de un cambio de paradigma es el poder esencial de un cambio considerable, ya se trate de un proceso instantáneo o lento y pausado
>
> Stephen Covey

Los lenguajes de programación pertenecen a uno o varios paradigmas de los que se especifican a continuación, se trata de una forma en como se resuelven los problemas de programación. En principio, todos los lenguajes de programación son *Turing-computables*, es decir que en cualquier lenguaje puedo describir cualquier algoritmo.

Lo particular de los paradigmas es que algunos de ellos simplifican el trabajo del programador, al poder realizar en menos líneas o más eficientemente algunas operaciones o programas.

Una de las decisiones más interesantes sobre elegir un lenguaje de programación para aprender a programar, es precisamente decidirlo en función de qué paradigma (o paradigmas) se quiere aprender.

En opinión del autor, el paradigma funcional (ojalá en un lenguaje sin tipo de datos) es una agradable puerta de entrada en el mundo de la programación porque ofrece un espacio donde se puede utilizar mucho de los conocimientos de secundaria (como lo son las funciones) y el programador se puede centrar en el diseño de las funciones per sé, dejando de lado características como la implementación de tipos y ciclos que pueden ser comprendidos luego, cuando se migre a otro paradigma.

A.1. Programación estructurada

El primer lenguaje de programación denominado FORTRAN (FORmula-TRANslator), perteneció a este paradigma, dando como origen a todos los lenguajes que tienen esta característica. Se trata de un paradigma que está muy ligado a la estructura física de la computadora, por lo que trabajar con variables, hacer ciclos (o loops) es muy natural.

El lenguaje, a juicio del autor, que es el máximo exponente de este paradigma es el lenguaje de programación C. En este paradigma todos los programas se modelan como procesos que modifican espacios de memoria dados y son ejecutados a partir de un punto inicial y de forma secuencial.

A.2. Programación funcional

Este es el segundo paradigma de programación, nace con el segundo lenguaje de programación de la historia, LISP en el año 1958 en el Instituto Tecnológico de Massachusets (MIT por sus siglas en inglés).

Este paradigma modela todos los problemas como funciones, donde de manera transparente, el manejo de la memoria queda como responsabilidad del intérprete o del compilador. Típicamente las listas son parte de los lenguajes y hay operaciones para ellas, lo que simplifica el desarrollo de aplicaciones que involucren estructuras de datos.

Tienen la característica de que la recursión es un recurso importante para poder ejecutar programas, ya sea usando recursión de pila o recursión de cola.

A.3. Programación orientada a objetos

Este es el tercer paradigma de programación, nace en 1965 con Modula y llega a su máxima expresión con Smalltalk en 1972, sin embargo, el paradigma logra popularidad con lenguajes híbridos, C++ y Java.

En este paradigma se tiene que los problemas se modelan como objetos, cada uno con un *estado* (conjunto de valores de sus campos) y con un *comportamiento* (conjunto de mensajes que recibe y que producen ya sea un cambio en el estado del objeto o que retornan un valor).

Es a través del diseño de varios objetos (comúnmente llamados clases) y su interacción, que se logra modelar los problemas. En particular esto es útil cuando se pueden reutilizar objetos en problemas diferentes, de modo que se ahorra la programación de ellas.

En opinión del autor, este no debería ser el paradigma en el cual se empiece a programar inicialmente. En caso de querer empezar a programar en este paradigma, es mejor usar un lenguaje como Java o C++,que tienen muchas características de lenguajes estructurados, las cuales pueden ser utilizadas para programar en ese paradigma.

A.4. Programación lógica

El lenguaje lógico por excelencia es Prolog, el cual nace en 1972 convirtiéndose en el cuarto gran paradigma de programación.

En este paradigma, los problemas se modelan como relaciones o condiciones lógicas entre elementos de conjuntos (conjuntos partida y solución) o como la aplicación de lo que se denomina *modus ponens* sobre conjuntos de datos. Aunque las relaciones como se ven en la sección 1.3 de la página 21, son meramente condiciones lógicas, normalmente las personas no estamos acostumbrados *a priori* a trabajar con este tipo de modelado.

Otro detalle, es que como en el paradigma funcional, las listas ya existen por defecto y también hay relaciones que permiten operarlas, simplificando el tiempo de trabajo para operar estructuras de datos.

A.5. Lenguajes multiparadigma

Existen lenguajes que tienen características de más de un paradigma, tal es el caso de Python o Ruby, que tienen caracteríticas de orientación a objetos, funcionales y estructurados. Por lo que se han convertido en lenguajes que se utilizan para aprender a programar en un gran número de universidades en el mundo y Python es de los lenguajes con mayor número de programadores en el mundo, por la simpleza de su código, aunque aún buscan como hacerlo más eficiente, pero los recursos de hardware que disponemos en la actualidad tienden a compensar el margen de tener que interpretar código escrito en Python.

Apéndice B

Ejemplos de lenguajes de programación

> Cuando alguien dice: 'quiero un lenguaje de programación en el que sólo tengo que decir lo que quiero hacer', le dan un chupetin
>
> Alan J. Perlis

> Un lenguaje de programación, con su sintaxis formal y las reglas de demostración que definen su semántca, es un sistema formal para el cual la ejecución del programa provee solamente un modelo
>
> Edsger Dijkstra

En esta sección se van a describir algunos lenguajes de programación y se harán algunos programas en cada lenguaje para explicar cómo se hace un programa sencillo en el lenguaje descrito.

Antes de iniciar con los lenguajes propiamente, se trabajará sobre un detalle que está en común con todos los lenguajes, que son los operadores aritméticos.

B.0.1. Operadores

A continuación se detallan los operadores de las operaciones básicas

1. Suma: +
2. Resta: -

3. Multiplicación: *

4. División: /

La tabla B.0.1 muestra los operadores lógicos (booleanos) separados por lenguaje. Esta tabla tiene resultados de *verdadero* y $falso$, o sea, son valores que se pueden categorizar en dos tipos y basta 1 bit.

	Scheme	**Erlang**	**C - Java**	**Prolog**	**Python**
$\wedge$	$(and\ b1\ b2\ldots)$	B1 and B2	P1 && P2	,	B1 and B2
$\vee$	$(or\ b1\ b2\ldots)$	B1 or B2	P1 || P2	;	B1 or B2
$<$	$(<\ p1\ p2\ \ldots)$	P1 < P2	P1 < P2	<	P1 < P2
$\leq$	$(<=\ p1\ p2\ldots)$	P1 =< P2	P1 <= P2	=<	P1 <= P2
$>$	$(>\ p1\ p2\ldots)$	P1 > P2	P1 > P2	>	P1 > P2
$\geq$	$(>=\ p1\ p2\ldots)$	P1 >= P2	P1 >= P2	>=	P1 >= P2

B.1. Scheme

Scheme se trata de un lenguaje de programación funcional que está basado en LISP. Se recomienda que utilicen el intérprete actual de este lenguaje que es *DrRacket*, el cual cuenta con versiones para Linux y Windows.

Scheme es un lenguaje que no es tipado, por lo que es más sencillo hacer una transcripción de una función matemática a este lenguaje.

La principal característia de los lenguajes funcionales es que toda la programación se hace mediante funciones, por lo que siempre existe algo llamado cálculo lambda (λ), que define el cuerpo de la función. Otra característica que tiene Scheme es que todo lo que debe ser ejecutado es de la siguiente forma (función $p_1 p_2 p_3$ ...), aunque la función sea aritmética, por lo que se considera que Scheme trabaja en $prefijo$, es decir, con la operación antes que los operandos.

Por ejemplo, consideremos la siguiente función:

$$f(n) = n + 1$$

En Scheme se vería así:

```
(define sucesor
   (lambda (n)
      (+ n 1)))
```

También se pueden hacer funciones por partes, utilizando la función *cond*, la cual recibe varias condiciones y ejecuta la primera que sea verdadera, por ejemplo, valor absoluto

```
(define valor-absoluto
   (lambda (n)
      (cond ((< n 0) (- n))
            (else n))))
```

B.1.1. Comentarios

Los comentarios en Scheme, son todas aquellas líneas que comiencen con punto y coma (;).

```
; esto es un comentario
```

B.1.2. Listas

La ventaja de Scheme, al ser basado en LISP (List Processing) es que permite trabajar con listas con mucha facilidad y cuenta con muchas funciones para agilizar la programación de listas. Una lista en scheme es: '(1 2 3 4).

Si por ejemplo, quisiera elevar al cuadrado todos los elementos de esa lista, me bastaría con una sóla línea:

```
(map (lambda (x) (* x x)) '(1 2 3 4))
```

Donde el map es una función que recibe otra función y en este caso, una lista (podría recibir más listas, si la función recibiera más parámetros pero todas las listas deben ser de la misma longitud), y el map se encarga de aplicar la función a cada elemento de la lista, retornando la lista con los resultados en el mismo orden que la entrada.

Luego para trabajar con una lista hay funciones constructoras como *append* o *cons* y hay funciones destructoras como *car* y *cdr*. Veamos lo que hacen en los siguientes ejemplos:

```
; retorna la cabeza de la lista
(car '(1 2 3 4))
1
;retorna la cola de la lista (la lista sin el primer elemento)
(cdr '(1 2 3 4))
'(2 3 4)

; Inserta un elemento al inicio de la lista
(cons 0 '(1 2 3 4))
'(0 1 2 3 4)
; Concatena dos (o más listas)
(append '(-1 0) '(1 2 3 4))
'(-1 0 1 2 3 4)
```

A continuación se muestran dos ejemplos que utilizan listas:

```
;; Función que calcula la longitud de una lista (cantidad de elementos
(define longitud
   (lambda (l)
      (cond ((null? l) 0) ;si es nula o vacía, retorna 0
            (else (+ 1 (longitud (cons l)))))))
```

```
;; Función que retorna en una lista la secuencia de números desde ini hasta fin
(define secuencia
   (lambda (ini fin)
      (cond ((< fin ini) '())
            (else (cons ini (secuencia (+ ini 1) fin))))))
```

En el apéndice C de la página 151 habrán más ejercicios, con código específico de acuerdo con los problemas. Si se utiliza el *DrRacket* para ejecutar basta con pulsar el botón *Ejecutar* o *Run* (Ctrl+R desde el teclado). Para ejecutar los programas, se invoca a la función desde la consola de trabajo (Ctrl+D desde el teclado).

B.2. Erlang

Este es un lenguaje funcional, al igual que Scheme, con la particularidad que tiene dos características adicionales, es un lenguaje que trabaja con *pattern matching*, lo que le permite colocar las condiciones en la declaración misma de la función. La otra particulardidad es que tiene *List comprenhension*, lo que permite potenciar el trabajo con listas, con mejores funciones de lo que hace Scheme.

Erlang trabaja con una máquina virtual y tiene compiladores para Windows y Linux por igual. Cuando se programa en Erlang, se debe escribir el código en un archivo cuyo nombre comienza con una letra minúscula, y con la extensión **erl**. Por ejemplo: $file.erl$. Este archivo debe tener como primera línea:

```
-module(file).
```

De este modo se declara el módulo $file$, para ejecutar una función programada en este archivo, se debe escribir el nombre del módulo, seguido de dos puntos y el nombre y parámetros de la función y termina la instrucción con un punto (‘.’). Por ejemplo:

```
file:fibo(10).
```

La segudna línea, debe indicar cuáles son las funciones públicas y el número de parámetros que estas reciben, por ejemplo:

```
-export([fibo/1]).
```

Donde estaríamos indicando que la función $fibo$ es pública y que recibe un parámetro.

B.2.1. *Pattern Matching*

El *Pattern Matching* consiste en poner condiciones que siguen el patrón indicado, entonces es un indicativo de que se cumple esa condición. El *patternmatching* se puede ejemplificar en el siguiente código:

```
fibo(0)->0;
fibo(1)->1;
fibo(N)->fibo(N-1)+fibo(N-2).
```

Como se puede apreciar, en el mismo código se indica si es 1 o 0 en lugar de hacer un condicional. Si esto no bastara, también existen los llamados *guards* como el que se muestra a continuación en un código equivalente al anterior.

```
fibo(N) when N < 2->N;
fibo(N)->fibo(N-1)+fibo(N-2).
```

B.2.2. Compilación

Para compilar el programa, se debe escribir $c(file)$. en la consola de Erlang y de esta manera se pueden ejecutar las funciones que se exportaron. Desde la consola de Erlang, para correr fibonacci, por ejemplo, se escribe:

```
file:fibo(10).
```

B.2.3. Comentarios

Los comentarios en el código de Erlang, se escriben con %

```
% Esto es un comentario
```

B.2.4. Listas

Las listas en Erlang se denotan con los paréntesis cuadrados ([]). Así una lista vacía es [] y una lista con los primeros 3 números naturales sería: [0,1,2].

Sin embargo, para maximizar el pattern matching, en Erlang se utiliza la notación $[Cabeza|Cola]$, así si hiciera por ejemplo:

```
[H|T] = [1,2,3,4].
% sería:  H = 1 y T = [2,3,4]

[Head|Tail] = [a].
% sería: Head = a y Tail = []

% seria un error [A|B] = []  porque no hay cabeza en una lista vacía.
```

Aprovechando el *pattern matching* veamos un par de funciones que utilizan listas (mismas que de la sección anterior con Scheme).

```
% Función que calcula la longitud de una lista
longitud([])->0;
longitud([_Head|Tail])->1+longitud(Tail).

% función que dado un inicio y un fin, retorna la lista desde inicio hasta fin
secuencia(Ini,Ini)->[Ini];
secuencia(Ini,Fin)->[Ini|secuencia(Ini+1,Fin)].
```

El guión bajo en la variable Head le indica al compilador que deliberadamente no voy a utilizar esa variable.

En la función secuencia, considere el caso base, cuando el inicio y el fin son iguales.

B.3. Prolog

De todos los paradigmas, el lógico es quizá el más atípico de todos, dentro de este paradigma se encuentra Prolog, que tiene una implementación de backtracking "incluída", pero se basa en el cumplimiento de condiciones lógicas, similares a la programación con restricciones.

B.3.1. Relaciones

Es primordial, para poder dar los primeros pasos en prolog, comprender qué significa una relación y qué representa desde el punto de vista lógico una expresión.

Veamos un ejemplo con $n!$, sea R una relación entre dos naturales, donde $aRb \Leftrightarrow b = a!$, pero, supongamos que tenemos que definir además qué significa ese símbolo de factorial también utilizando condiciones lógicas.

En ese caso, debemos indicar que $0R1$, como un caso base, luego podemos definir los siguientes elementos: $aRb \Leftrightarrow \exists a' = a - 1 \wedge b'/a'Rb' \wedge b = b' \times a$ O dicho textualmente: para que b sea el factorial de a, quiere decir que existe un a prima que es igual que a menos 1 y ese número también tiene un factorial, al que llamamos b prima. Luego, b es b prima por a.

Esta consciencia es la que debemos despertar cuando nos enfrentamos a definir programas en Prolog, porque todo son relaciones lógicas.

B.3.2. Sintaxis, programagprolog básico, cut y pattern matching

Con la esperanza que la lógica haya sido comprendida, es hora de visualizar un primer programa en prolog, iniciaremos con el mismo factorial descrito en la sección anterior. Es importante resaltar que una relación en prolog no retorna nada más que *yes* o *no*, pero intenta encontrar aquellos valores que hacen que la relación retorne *yes*

```
fact(0,1).
fact(A,B):- A1 is A-1, fact(A1,B1), B is B1*A.
```

Primer detalle, las mayúsculas indican valores sin anclar, es decir, sin asignar, esta asignación ocurre durante la ejecución.

También es importante señalar que Prolog funciona con backtracking, visto en la sección 8.7 en la página 128, donde cada una de las proposiciones (en el ejemplo, definiciones de $fact$) son los "hijos" de los nodos. Para indicarle al compilador que a partir de cierto hijo o de cierta condición, ya no se debe seguir

revisando los hijos (es decir, no hay que meter más en la pila, se utiliza el *cut* denotado por !. Así en el mismo código de factorial, podemos evitar que siga al segundo caso, indicando que luego de $O! = 1$ de esta manera:

```
fact(0,1):-!.
fact(A,B):- A1 is A-1, fact(A1,B1), B is B1*A.
```

También podemos ver como en el caso de $0! = 1$ no hay ningún *if* ni notación de condicional de algún tipo, esto ocurre por el *pattern matching*, el cual consiste en que si la entrada de la relación calza (hace *match*) con lo indicado, entonces es un indicativo de que esa relación es la indicada.

Para ello vamos a ver otro ejemplo con fibonacci (con recursión de pila y muy ineficiente, pero sólo para ilustrar el concepto):

```
fibo(0,0):-!.
fibo(1,1):-!.
fibo(N,Fn):- N1 is N-1, N2 is N-1,
             fibo(N1,Fn1), fibo(N2,Fn2),
             Fn is Fn1+Fn2.
```

B.3.3. Caso con listas

Prolog comparte la notación de listas de Erlang (de hecho este último copió la sintaxis de prolog y el *pattern matching*).

Por lo que podemos ejemplificar las funciones *secuencia* y *longitud* una vez más pero en prolog. (Vistas en Erlang y en Scheme)

```
longitud([_H|T], X):- longitud(T,X1), X is X1+1
longitud([],0).

secuencia(A,A,[A]):-!.
secuencia(I,F,[I|Sec]):- I1 is I+1, secuencia(I1,F,Sec).
```

B.4. C

C es el lenguaje más importante de los últimos tiempos, se trata de un lenguaje que ha sido base de muchos otros y cuyas buenas caraceristicas se han reproducido muchas veces. Sin embargo, tiene muchas condiciones necesarias para poder hacer un programa.

B.4.1. Tipos de datos

En C, existen conceptos como variables que permiten almacenar valores y cambiar su valor dependiendo del momento de la ejecución. Estas variables tienen un tipo, que se puede catalogar de la siguiente forma:

- Enteros:

1. char (con capacidad de 1 byte)
2. short int (con capacidad de la mitad de un entero)
3. int (entero, con un tamaño de una palabra)
4. long int (con el doble del tamaño de un entero)

- Punteros

  Un puntero es una dirección de memoria, también se utilizan como arreglos, es decir espacios de memora estáticos que pueden ser utilizados para almacenar conjuntos de datos, que pueden ser de cualquiera de los tipos posibles en C o puede incluso ser de tipos creados dentro del lenguaje.

- Flotantes

  1. float (con una capacidad de mantisa y exponente, dada)
  2. double (con el doble del flotante)
  3. long double (con el doble del double).

- void (sin tipo, normalmente para métodos sin valor de retorno)

B.4.2. El main

En C, para poder correr un programa, debe crearse un método denominado main. El cual será lo único que se ejecutará. Así si se quisiera programar algo más complejo, se llaman las funciones principales desde el main. Para poder visualizar los valores, se debe imprimir en pantalla lo que se quiere.

Ejemplo de un código para fibonacci en C:

```
#include <stdio.h>

int fibo (int n){
    if(n == 0 || n ==1)
        return n;
    return fibo(n-1)+fibo(n-2);
}

int main ()
{
     printf("%d\n",fibo(10));
}
```

La función printf, es de la biblioteca stdio.h, por ello es que se agrega al inicio del programa. el %d, es para indicar que va a imprimir un número en decimal y el \n es para realizar un cambio de línea al finallizar el programa. Luego, cada % es reemplazado secuencialmente por los valores que arrojen esas funciones (si hubiesen n %, entonces deben haber n parámetros luego del string, separados todos por coma).

B.4.3. Comentarios

Los comentarios en el código de C, se escriben de dos formas, comentarios de línea o de bloque.

```
// Esto es un comentario de línea

/* Esto
   es uno de
   bloque */
```

B.5. Java

Java tiene varias particularidades, por un lado se dice que es un lenguaje orientado a objetos, sin embargo comparte gran parte de su sintaxis con C y C++, de modo que es relativamente fácil pasar entre estos lenguajes. Quizá la principal diferencia es que Java corre sobre una máquina virtual y el manejo de memoria es directamente responsabilidad de esta máquina virtual, de modo que se cuenta con un recolector de basura, que todas aquellas estructuras que pierden referencia, se borran de memoria directamente. Esta característica, hace que sea poco recomendable para sistemas de tiempo real, caso contrario ocurre con C.

B.5.1. Declaración de clases

En Java es muy aconsejable tener una clase por archivo y que ésta sea pública (eso quiere decir que el archivo tiene que tener el mismo nombre que la clase, con la extensión *.java*).

Partes de una clase

Una clase se divide en dos partes, por un lado están los campos, que son los datos (espacios de memoria del objeto) que van a tener y por otro los métodos. Algunos de estos métodos, son atributos de la clase. Es importante notar que no es necesario tener en un campo cada posible atributo. Por ejemplo, si se tiene una clase círculo, sus atributos serían: radio, puntoCentral, área, circunferencia. Sin embargo, sólo necesito guardar un campo (puede ser el radio), dado que los demás atributos se pueden calcular a partir de él[1].

Para proteger los campos y los métodos de usos no autorizados, se definen los siguientes niveles de seguridad de ellos:

1. *private*: Sólo dentro de la clase se puede ver o modificar este campo o método (muy recomendado para campos y aquellos métodos internos de la clase)

[1] Nótese que también se puede guardar la circunferencia y con ello despejar los otros atributos

2. *protected*: Sólo dentro de las clases que se encuentran en el mismo paquete se puede ver o modificar este campo o método.

3. *public*: Cualquier clase puede tener acceso a este método o campo. Es muy necesario para que los objetos interactúen con otros objetos, que es el fin de la POO.

B.5.2. Tipos

Los tipos son los mismos que tiene C, sin embargo, existe además el objeto nativo *String* que permite sumar strings de texto para concatenarlas, es decir formar un solo String que contenga como primera parte el primer String y como segudna parte, el segundo String.

Además la clase String cuenta con una serie de métodos que permiten hacer varias operaciones, como poner un segmento en mayúsculas, o en minúsculas,obtener el caracter de una posición específica, saber el largo del String, entre muchos otros. Esto se puede consultar en el API de java que se encuentra online y que resulta muy útil tenerlo a mano para hacer programas utilizando las bibliotecas[2] que en Java se denominan paquetes.

B.5.3. El constructor

Las clases tienen métodos que se denominan ***constructores***, los cuales son los responsables de iniciar los campos de la clase y reservar el espacio en memoria para el objeto.

Este constructor no tiene otro tipo que el mismo nombre de la clase y normalmente recibe los valores iniciales de aquellos campos que se buscan empezar.

B.5.4. Ejemplo de programa pequeño: Clase Punto2D y Clase Recta

En esta sección, se crearán dos clases que se relacionan entre ellas y que explica como es que los objetos se relacionan entre ambos.

Se recomienda siempre empezar a programar con objetos matemáticos porque sus comportamientos y campos están bien definidos y pueden reemplearse en el cualquier contexto. Además, se recuerda que ambas clases deben estar en un archivo que se llame igual que la clase.

```
public class Punto2D{

    double x;
    dobule y;
//Constructor (valor por defecto, el origen)
public Punto2D()
```

[2]Muchas personas utilizan el término *librerías*, que viene a ser una mala traducción del término en inglés *library*

```
{x = y = 0.0;}

public punto2D(double valX, double valY)
{x = valX;   y = valY;}

void mover(double Dx, double Dy){
x = x+Dx;
y = y+Dy;
}

boolean equals (Punto2D pto){
return this.x == pto.getX() && this.y == pto.getY();
}

public double getX(){return x;}
public double getY(){return y;}

}


public class Recta{

double m;
double b;

//Constructor
public Recta(double m, double y){
this.m = m;
this.y = y;
}

Punto2D getPunto(double x){
return new Punto(x, m*x+b);
}

boolean contiene(Punto2D pto){
return getPunto(pto.getX()).equals(pto);
}

}
```

Como se puede apreciar, una recta, puede retornar un punto 2d, para un valor "x.^{en} particular, que se encuentre sobre la recta o indicar si un punto está sobre una recta. Se pueden hacer con esos mismos campos, muchos métodos tales como la recta perpendicular a un punto dado de la recta original (Se sugiere

hacer el ejercicio dentro de esas clases).

B.5.5. El main

Al igual que en C, todo lo que se va ejecutar en el programa debe de hacerse bajo el main, que dado que se trata de un objeto, es un método que tiene que llamarse igual:

```
public static void main(String [] args){ ... }
```

Para poder probar el código de las clases, es neceario crear un Main y hacer programas pequeños que sean los responsables de ingresar datos y probar los resultados.

B.6. Python3

Python se ha convertido en el lenguaje de programación que a nivel mundial más universidades lo utilizan para aprender a programar. Se trata de un lenguaje de Script, por lo que se ejecuta desde la primera línea hasta la última por medio de un intérprete. Tiene bibliotecas muy poderosas para diversas funciones y es sencillo para el programador aprender a escribir su código. En esta sección se va a dejar por fuera el uso de POO en python y se va a enfocar más en un estilo funcional (sin utilizar lambda).

Python es un lenguaje de scripting, lo que quiere decir que no requiere forzosamente un main ni tampoco ser compilado. Puede ser ejecutado desde diversas terminales y existen dos versiones principales de bibliotecas para ptyhon, en esta sección trabajaremos con Python3 de forma estándar.

Como se indicó en el apéndice A Python es un lenguaje multiparadigma, por lo que los programas se pueden resolver también en cada uno de los diferentes paradigmas, con sus ventajas y desventajas.

B.6.1. Definición de funciones

Las funciones pueden ser recursivas (como lo requieren Scheme, Erlang o Prolog) o pueden ser iterativas (como sería en C o Java), ninguna en particular es mejor o peor que otra a priori. Depende de cómo quiera trabajarlo y de otras consideracioens que probablemente con la práctica se verán.

Otro detalle importante en python es que la identación importa, la forma en como defino que algo está dentro de la definición de una función o dentro de un ciclo, es porque se encuentra más identado (con tabs o con igual número de espacios) que su predecesor.

Veamos un primer programa en python:

```
def sucesor(n):
     return n+1
```

Como se puede apreciar, para definir la función, se utiliza la palabra reservada def, luego, no se debe indicar qué tipo tiene n, [3] y la función finaliza con la palabra reservada *return* que retorna el valor $n+1$. Es importante notar que la palabra *return* se encuentra un tab de separación del def, indicando que ese return pertenece a la definición anterior.

Veamos un ejemplo con más líneas:

```
1 def fibo(n):
2      if n == 0 or n == 1:
3           return n
4      return fibo(n-1)+fibo(n-2)
```

Veamos este otro ejemplo, como se ve en la línea 1 se define la función $fibo$, las demás líneas están una identación a la derecha de esa definición. En la línea 2 se encuentra un condicional, que pregunta si n es cero o si n es 1. Si la condición fuera verdadera, entonces se ingresa a esa nueva estructura de control (el if). La línea 3 sólo corre cuando la condición presentada, resulte verdadera. Importante mencionar que *return* finaliza la ejecución, por lo que si se ejecuta la línea 3, la línea 4 no se ejecutaría. Finalmente, la línea 4 se ejecuta haciendo una invocación a la llamada recursiva de fibonacci (una vez más, con recursión de pila).

B.6.2. Comentarios

Al igual que en C o Java, existen dos tipos de comentarios, los comentarios de línea, que son precedidos por el símbolo numeral o los comentarios de bloque, que son iniciados y finalizados por tres comillas.

Ejemplo:

```
# Esto es un comentario de línea

'''
Esto es un comentario de bloque
Puede tener tantas líneas como se desee
Finaliza al poner de nuevo, tres comillas simples
'''
```

B.6.3. Listas

Al igual que en Erlang o Prolog, las listas en Python se denotan por []. Pero tienen una particularidad, se pueden accesar por medio de la posición que ocupan en la lista, o mejor dicho, por el número de corrimientos que debe desplazarse desde la dirección en memoria que tiene la lista, así la primera posición requiere un desplazamiento de cero.

Por ejemplo:

[3] Nótese que python utiliza tipado dinámico a lo interno

```
Lista = [5,10,15]
# Donde L[0] = 5 , L[1] = 10 y L[2] = 15
```

Otra particularidad que tienen las listas en Python es que se pueden "cortar" utilizando los dos puntos (:), desde una posición i (incluida) hasta una posición j (sin incluir), de la forma $L[i : j]$. Si alguno de los dos lados alrededor de los dos puntos se dejara en blanco, implicaría hasta al final o desde el inicio por ejemplo:

```
Lista = [5,10,15,20]
L2 = Lista[1:]
L3 = Lista[:2]
L4 = Lista[1:2]
# Eso haría L2 = [10,15,20]
# L3 = [5,10]
# L4 = [5]
```

Apéndice C

Algunas soluciones

Hablar es barato. Enséñame el código

Linus Torvalds

Rendirse es la manera más dolorosa de 'resolver' un problema

Anónimo

Este apéndice busca servir de guía y ejemplo de como resolver algunos ejercicios planteados en este libro. No busca ser exhaustivo porque se desea que los estudiantes puedan también divertirse haciendo sus ejercicios, donde se espera que cuenten con el apoyo de un profesor que les corrija o les sepa dar pistas sobre cómo resolver algún ejercicio si es que tuvieran algún problema con alguno.

También se le agradece al lector considerar que en algunos lenguajes, la función dada debe ser invocada desde una función ejecutable superior (como en el main en el caso de C o Java) y que no se muestran programas completos.

Los ejercicios que se van a resolver en todos los lenguajes son de la sección 3.7.3 de la página 61 el 4, 7 y 11 respectivamente.

Se invita a no revisar esta sección hasta haber agotado muchos intentos de resolver los problemas, la idea del epígrafe, consiste en no rendirse, sino en verdaderamente interiorizar como se resuelve cada problema.

C.1. Scheme

En esta sección se van a utilizar las siguientes funciones:

- even?

  Indica verdadero si el número es par y falso en caso contrario.

- odd?

  Indica verdadero si el número es impar y falso en caso contrario.

- zero?

  Indica verdadero si el número es cero y falso en caso contrario.

- remainder

  Recibe dos parámetros, retorna el residuo de dividir el primero por el segundo

- quotient

  Recibe dos parámetros, retorna el cociente de dividir el primero por el segundo

- expt

  recibe dos parámetros, a y n, retorna a^n

```
; Indica si n es primo retorna verdadero (#t) o si no lo fuera, falso (#f)
(define primo?
     (lambda(n)
          (cond ((= n 2) #t)
                ((even? n) #f)
                (else (primo?-aux n 3)))))

(define primo?-aux
     (lambda (n i)
          (cond ((> (* i i) n) #t)
                ((zero? (remainder n i)) #f)
                (else (primo?-aux n (+ i 2))))))

; Invertir un número
(define invertir
     (lambda (n)
          (invertir-aux n 0)))

(define invertir-aux
     (lambda (n inv)
          (cond ((zero? n) inv)
                (else (invertir-aux (quotient n 10)
                                    (+ (remainder n 10)
                                       (* 10 inv)))))))
```

Función equivalente a: $\prod_{i=0}^{n} \frac{1}{2^i}$

```
(define producto-fracciones
     (lambda (n)
          (cond ((zero? n) 1)
                (else (* (/ 1 (expt 2 n))
                         (producto-fracciones (- n 1)))))))
```

C.2. Erlang

En esta sección se utilizan las siguientes funciones primitivas:

- rem

  Es un operador binario que retorna el residuo del primer operando con el segundo operando (ambos deben ser enteros).

- div

  Es un operador binario que retorna el cociente del primer operando con el segundo operando (ambos deben ser enteros).

- pow

  Del módulo de math, eleva el primer parámetro a la potencia indicada en el segundo.

```
% Función que indica si un número es primo o no
primo(2)->true;
primo(N)when N rem 2 == 0->false;
primo(N)->primo(N,3).

primo(N,I)when I*I > N->true;
primo(N,I)when N rem I == 0->false;
primo(N,I)->primo(N,I+2).

% Función para invertir un número
invertir(N)->invertir(N,0).

invertir(0,Inv)->Inv;
invertir(N,Inv)->invertir(N div 10, Inv*10+ N rem 10).
```

Función equivalente a: $\prod_{i=0}^{n} \frac{1}{2^i}$

```
sumaFracciones(0)->1;
sumaFracciones(N)->(1/(math:pow(2,N))) * sumaFracciones(N-1).
```

C.3. Prolog

Primitivas utilizadas en estos programas:

- rem

  El rem es residuo de dos números

- \+

  Este es el not en gprolog

```
%Relación que retorna yes si N es primo, no en caso contrario.
primo(2):-!.
primo(N):- 1 is N rem 2, primo(N,3).

primo(N,I):- I*I > N,!.
primo(N,I):- \+ 0 is N rem I, I2 is I+2, primo(N,I2).

invertir(N,I):-invertir(N,0,I).

invertir(0,I,I):-!.
invertir(N,In,I):- N1 is N div 10, In2 is In*10+N div 10, invertir(N1,In2,I).
```

Función equivalente a: $\prod_{i=0}^{n} \frac{1}{2^i}$

```
sumaFracciones(N,I):-sumaFracciones(N,1,I).

sumaFracciones(0,I,I):-!.
sumaFracciones(N,T,I):- X is 2**N, T2 is T*(1/X), N1 is N-1,
                        sumaFracciones(N1,T2,I).
```

C.4. C

```
int primo(int n){
     if(n == 2) return 1;
     if(n%2 == 0) return 0;
     int x = 3;
     while(x*x <= n){
        if(!(n%x)) return 0;
        x+=2;
     }
     return 1;
}

int invertir(int n){
    int resp = 0;
    while(n){
```

```
        resp*=10;
        resp+=n%10;
        n/=10;
    }
    return resp;
}
```

Función equivalente a: $\prod_{i=0}^{n} \frac{1}{2^i}$

```
long double sumaFracciones(int n){
    long double resp = 1.0;
    for(int i = 0; i<=n; i++){
        resp*= 1/(1<<i);
    }
    return resp;
}
```

C.5. Java

```
public boolean primo(int n){
     if(n == 2) return true;
     if(n%2 == 0) return false;
     int x = 3;
     while(x*x <= n){
        if(!(n%x)) return false;
        x+=2;
     }
     return true;
}

public int invertir(int n){
    int resp = 0;
    while(n>0){
        resp*=10;
        resp+=n%10;
        n/=10;
    }
    return resp;
}
```

Función equivalente a: $\prod_{i=0}^{n} \frac{1}{2^i}$

```
public double sumaFracciones(int n){
    double resp = 1.0;
    for(int i = 0; i<=n; i++){
```

```
        resp*= 1/(1<<i);
    }
    return resp;
}
```

C.6. Python

Python se puede utilizar tanto recursiva como iterativamente, las soluciones no son únicas, si el lector gusta, intentar la otra versión, queda más que invitado a hacerlo.

```
# Retorna True si es primo y False si no
def primo(n):
     if n == 2:
         return True
     if n%2 == 0:
         return False
     return primoAux(n,3)

def primoAux(n,i):
     if i*i > n:
      return True
     if n%i == 0:
        return False
     return primoAux(n,i+2)


def invertir(n):
     resp = 0
     while n:
          resp*=10
          resp+=n%10
          n//=10
     return resp
```

Función equivalente a: $\prod_{i=0}^{n} \frac{1}{2^i}$

```
def sumaFraciones(n):
     resp = 1
     for i in range(n+1):
          resp*= 1/(2**i)
     return resp
```